High Performance Computing using Infiniband-based clusters

ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Control and Computer Engineering (31^{th}cycle)

High Performance Computing using InfiniBand-based clusters

By

Masoud Hemmatpour

******

Supervisor(s):
Prof. Bartolomeo Montrucchio

Doctoral Examination Committee:
Prof. Maurizio Rebaudengo, Politecnico di Torino
Prof. Giancarlo Iannizzotto, Universitá degli Studi di Messina
Prof. Sorin Moraru, University of Brasov
Prof. Davide Quaglia, Universitá di Verona
Prof. Claudio Zunino, Istituto di Elettronica e Ingegneria dell'Informazione e delle Telecomunicazioni (IEIIT)

Politecnico di Torino
2019

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Masoud Hemmatpour
2019

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

I would like to dedicate this thesis to my mother who is priceless in my life.

Acknowledgements

First and foremost, I would thank my outstanding advisor Prof. Bartolomeo Montrucchio for giving me all the research freedom and support. Furthermore, a great thank to Prof. Maurizio Rebaudengo for his outstanding help. Then, I would carry on with a list of great friends in all over the world, specially those who had tons of patience for various reasons. A special thanks, of course, would go to the thesis jury members for taking their time to thoroughly review the thesis. Also, I would like to thank Prof. Sadoghi for all his support in my thesis. Then, I would express my deepest gratitude to my beloved family that has always been there as an eternal source of inspiration, support, and understanding. Most importantly, thank to my mother for teaching me the real values of life. I hope my life and my choices have honored her. Last but not least, an special thank to Mahsa for her unconditional support. But despite the urge to elaborate on my thanks, the next few lines will be somewhat different. They will summarize some important lessons I have learned during these three years, which is what I am most thankful for. When this journey started, my belief was that doing a doctorate means proposing totally a novel method in my project. But it turned out that shiny and polished ideas are the outcome of hardworking for a long period of time and one of the things that doing a PhD teaches me is that, just I should work hard and be optimistic maybe the reward is just around the corner!

Abstract

As demand for big data analytics grows every day, companies have become aware of the critical role of real-time data-driven decision making to gain a competitive edge. This creates a challenge for companies needing to accelerate (fine-grained) access to massively distributed data, in particular, those dealing with online services. NoSQL systems, in particular distributed in-memory key-value stores, are vital in accelerating memory-centric and disk-centric distributed systems.

A *distributed key-value* store offers a flexible data model with more performant but weaker consistency to partition data across many nodes on a computer network. Starting in mid-2000, numerous commercial key-value stores have emerged, each with its own unique characteristics, such as Google Bigtable, Amazon Dynamo, and Facebook Cassandra to enable managing massively distributed data at unseen scale, which simply was not feasible with traditional relational database systems running on commodity hardware. These systems have become critical for large-scale applications, such as social networks, realtime processing, and recommendation engines to achieve higher performance.

Distributed systems are commonly built under the assumption that the network is the primary bottleneck, however this assumption no longer holds by emerging high-performance protocols in datacenters. Designing distributed applications over such protocols requires a fundamental rethinking in communication components in comparison with traditional protocols (i.e., TCP/IP). Much research has been carried out in order to improve the communication performance either by optimizing the existing protocol or inventing new communication standards.

A great deal of work on high-performance communication has led to modern high-speed networks including InfiniBand, RoCE, and iWARP, which support Remote Direct Memory Access (RDMA). RDMA blurs the boundary of each machine by creating a virtual distributed, shared memory among connected nodes, i.e., sub-

stantially reducing communication and processing on the host machine. Through RDMAs, clients can now directly access remote memory without the need to invoke the NoSQL's traditional client-server model. This motivates the NoSQL community to invest in developing purely in-memory key-value stores with RDMA capability, such as HydraDB, Herd, Pilaf, DrTM, FaRM. RDMA capable protocol (i.e., InfiniBand) supports legacy socket applications through *IP over InfiniBand* (IPOIB); however, running existing in-memory systems on top of it can not efficiently exploit the benefits in the infrastructure. So existing in-memory key-value stores strive to reduce latency and achieve higher performance by exploiting RDMA operations. In this thesis, commonly used underlying structure and data concurrency in RDMA-enabled in-memory key-value store are discussed. Furthermore, performance challenges of the RDMA operations have been investigated. State of the art are reviewed and evaluated based on the achieved knowledge on the RDMA operation. Finally, a novel in-memory key-value store is presented and evaluated in comparison with the state of the art.

Riassunto

L'analisi dei big data svolge un ruolo fondamentale negli ultimi anni e molte aziende hanno preso coscienza dei vantaggi competitivi che le decisioni real-time data-driven offrono. Ciò crea una sfida per le aziende che hanno bisogno di accelerare l'accesso (a grana fine) a enormi quantità di dati distribuiti, in particolare per coloro che si occupano di servizi online. I paradigmi NoSQL, in particolare distributed in-memory key-value stores, sono fondamentali per velocizzare i sistemi distribuiti del tipo memory-centric e disk-centric. Il distributed key-value store offre un modello di dati flessibile più efficiente ma meno consistente. Tale modello risulta più debole nel partizionare i dati su molti nodi di una rete di computer. A partire dalla metà degli anni 2000, sono emersi numerosi key-value stores commerciali, ciascuno con caratteristiche uniche, come ad esempio Google Bigtable, Amazon Dynamo, and Facebook Cassandra. Essi consentono la gestione di una enorme quantità di dati su scala invisibile, che semplicemente non era possibile con i tradizionali sistemi basati su database relazionali eseguiti su hardware. Questi sistemi sono diventati di vitale importanza per le applicazioni su larga scala, come i social network o il trattamento in tempo reale dei dati, al fine di ottenere prestazioni più elevate. I sistemi distribuiti sono comunemente creati assumendo che la rete rappresenti il collo di bottiglia principale. Tuttavia questa ipotesi non è più valida a causa di protocolli ad alte prestazioni emergenti nei data center. La progettazione di applicazioni distribuite su tali protocolli richiede un ripensamento fondamentale delle componenti di comunicazione rispetto ai protocolli tradizionali (cioè TCP/IP). Molte ricerche sono state condotte, ottimizzando i protocolli esistenti o definendo nuovi standard di comunicazione, al fine di migliorare tali prestazioni comunicative. La grande quantità di lavoro svolta sulle comunicazioni high-performance ha portato alle moderne reti ad alta velocità, tra cui InfiniBand, RoCE e iWARP, che supportano il Remote Direct Memory Access (RDMA). RDMA crea una memoria virtuale distribuita e condivisa tra i nodi connessi e ciò riduce, sostanzialmente, la comunicazione e

l'elaborazione dei dati sulle macchine. Tramite gli RDMA, i client possono ora accedere direttamente alla memoria remota senza dover richiamare il tradizionale modello client-server di NoSQL. Questo ultimo aspetto motiva la comunità NoSQL a investire nello sviluppo di in-memory key-value stores puri con capacità RDMA, come HydraDB, Herd, Pilaf, DrTM, FaRM. I protocolli compatibili con RDMA, per esempio InfiniBand, supportano le applicazioni basate sui socket legacy tramite IP over InfiniBand (IPOIB); tuttavia, l'esecuzione di in-memory system esistenti su IPOIB, non possono sfruttare in modo efficiente i vantaggi dell'infrastruttura. Quindi, i sistemi in-memory key value stores si sforzano di ridurre la latenza e ottenere prestazioni più elevate sfruttando le operazioni RDMA. In questa tesi vengono discussi la struttura sottostante e la concorrenza negli in-memory key-value store con RDMA-enabled prima discussi. Inoltre, sono state studiate le sfide prestazionali delle operazioni RDMA. Lo stato dell'arte è stato analizzato e valutato sulla base delle conoscenze acquisite sulle operazioni con RDMA. Infine un nuovo in-memory key-value store viene presentato e confrontato con lo stato dell'arte.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The challenge to accelerate fine-grained access to massively distributed data is a hot debate these years. The quick access to data is enabled out of essential services and hardware technologies. KEY-VALUE STORE is a typical solution for quick access to data. Key-value store uses an associative array as the fundamental data model where each key is associated with one and only one value in a collection to store and to retrieve data. Application of key-value store is not limited to sciences, it can be exploited in highly complex computing environments. Surprisingly, key-value store is not only becoming more affordable, but also its application beyond the sciences is broadening to include numerous business applications. It can deliver significant value if company is looking to boost the frequency and speed of calculations and analysis of data with much higher accuracy.

The first generation of key-value stores first turned to storage-class memory such as solid-state disks (SSDs) to achieve higher performance. This approach focused on exploiting SSDs as a cache between main memory and disks. Subsequently, as the size of main memory increases and the technology to assemble a large shared memory space among a set of machines becomes cheaper, there is a rising interest to develop purely in-memory key-value stores. However, scaling an in-memory key-value store was challenging with *Remote Procedure Call* (*RPC*) over traditional Ethernet due to the network overheads and it requires scalable high-performance interconnect infrastructure. High performance computing innovations provide larger and faster access to the shared data. One of the fastest and most efficient interconnect solution is InfiniBand adapters. The applications that run on top InfiniBand adapter are managed

the same as applications running on Ethernet Network Interface Card (NIC). One of the significant features of the Infiniband adapters is supporting remote Direct Memory Access (RDMA). The RDMA enables to access the memory of remote machine bypassing the operating system and CPU processing of the remote machine which significantly reduce the latency of process. In-memory key-value stores embrace RDMA technology in their design in order to achieve higher performance, emerging the RDMA-enabled in-memory key-value store.

1.1 Problem statement

Network bandwidth is not a bottleneck in RDMA-based in-memory key-value stores since they deal with small message sizes. However, new RDMA-based in-memory stores strive to come up with a novel design which differs in the memory management, choice of the RDMA operations and the connection types to saturate the message rate. Although the existing RDMA-enabled in-memory key-value stores perform well, in some aspects these designs are not well-suited. This thesis strives to crystal clear the challenges of designing a high-performance in-memory key-value store and propose methods to overcome the bottlenecks.

1.2 Research Contributions

In this section, the key research contributions that address the aforementioned problem are described.

This thesis presents one-of-a-kind comprehensive study of modern RDMA-based in-memory key-value systems including HydraDB [1], Pilaf [2], HERD [3], FaRM [4], and DrTM [5] as well as well-known legacy in-memory systems such as Memcached [6] and Redis [7]. Modern key-value stores are illustrated in an unified representation to show architectural differences along with strengths and weaknesses of each system. The key performance challenges of how to exploit RDMA in in-memory key-value stores are reviewed and a comprehensive evaluation methodology and extensive analysis of existing systems are presented.

Our key evaluation findings are summarized as follows:

- Exploiting one-sided RDMA-operation (i.e., `WRITE`) in exchanging the request/response is not always the best choice
- the performance distribution of memory access is greatly influenced by when the data is clustered or unclustered, and the clustered access achieves higher performance
- RDMA-based systems can serve request in an uniform way comparing to the traditional TCP systems
- the latency of the legacy systems are up to two order of magnitude higher than the RDMA-based systems
- systems using inline RDMA message are more sensitive to the size of message comparing to non-inline RDMA message
- increasing the size of the value will decrease the performance.

Finally, through exploiting the knowledge of analysing the state of the art, Kanzi, a distributed RDMA-enabled in-memory key-value store is proposed in this thesis.

1.3 Thesis structure

As illustrated in Figure 1.1, this thesis is divided into 8 chapters, as follows: Chapter 2 presents movement to NOSQL systems and preliminaries of the RDMA operations; Chapter 3 describes key-value stores as well as exploited underlying data structures and concurrency controls. Chapter 4 provides an overview of the state of the art which is part of this thesis, focusing on indexing, concurrency control, and RDMA operations. Chapter 5 contains the main performance challenges of the RDMA operations. In this chapter, different components such as memory, hostbus communication, network adaptor memory, RDMA features, network communication, and application level issues are investigated. Chapter 6 describes Kanzi and RDMA-enabled in-memory key-value store. Chapter 7 describes experimental evaluation of the performance challenges in RDMA operations as well as the results achieved on Kanzi. Finally, Chapter 8 draws some conclusions, and outlines the future research activities.

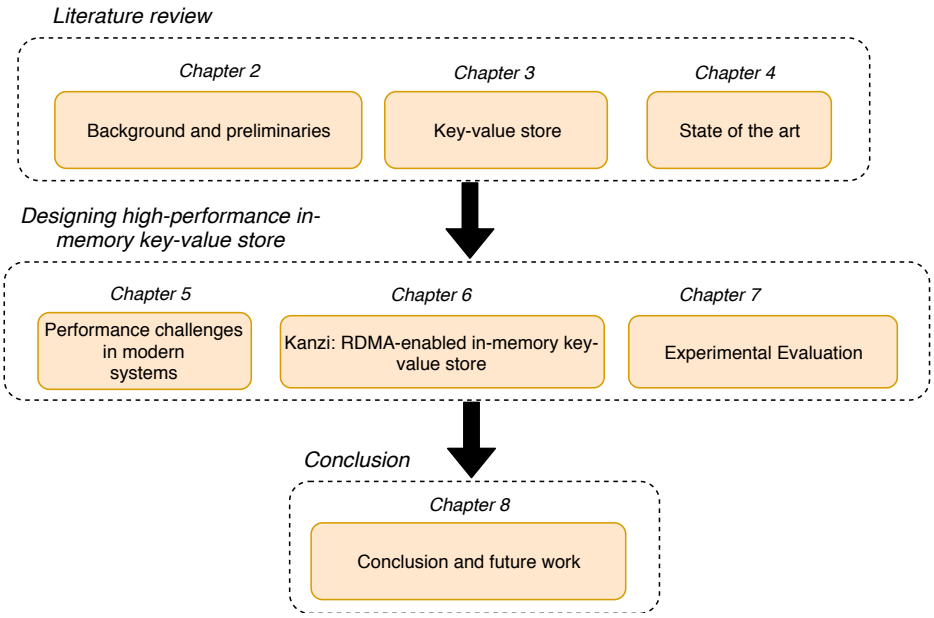

Fig. 1.1 Thesis structure.

Chapter 2

Background and preliminaries

In this chapter, motivations and main drivers of the NoSQL movement will be briefly described along with a classification and characterization of NoSQL databases. Then, the Remote Direct Memory Access (RDMA) in high-performance protocols will be discussed.

2.1 NOSQL Database Motives

Although, traditional databases with relational data model provide various features such as strict data consistency, these features might be more than necessary for several use cases. Moreover, with the development of the Internet, the traditional relational databases faced many challenges such as efficient big data storage and access. NoSQL (Not Only SQL) databases emerged to provide high scalability in storing and accessing big data. NoSQL is the another type of data storage that is used to store huge amount of data in high-demand environment. NoSQL databases can be classified to four categories according to the types of data: column-oriented databases, document databases, graph databases, and key-value databases.

Column-oriented databases are designed to store data in columns, unlike relational databases where data are stored in contiguous rows. This change in storage design results a better performance. For example, in row-oriented databases all columns of rows that satisfies a query are retrieved which cause unnecessary input/output, however column-oriented databases only retrieve the required columns. Furthermore, column-oriented databases can be stored efficiently since data-type and

range of values are fixed for each column which can help to compress data. On the contrary, row-oriented storage deals with multiple data types for different columns on each row.

Since in column-oriented databases columns share nothing, parallel actions can be performed on different columns. This feature enables columnar databases to be compatible with MapReduce framework, which speeds up processing of large amount of data by distributing the query on large number of systems [8]. Popular column-oriented databases are Hypertable [9], HBase [10], HadoopDB [11], and Cassandra [12].

Document-oriented databases store any format of documents such as XML (eXtensible Markup Language) [13] and JSON (JavaScript Object Notation) [14] as a record in the database, which can be addressed via a unique key. Since different documents can have different number and type of fields, these databases provide more flexibility in comparison with relational databases. Furthermore, this feature allows document-oriented databases to store data efficiently by avoiding empty fields in different documents. Practical usability of these databases are in content management systems, mobiles, gaming and archiving. The widely used document-oriented databases are MongoDB [15], CouchDB [16], and RavenDB [17].

Graph databases represent data as a network structure containing nodes and edges. Nodes and edges have their own properties to describe the real data with their relationships. Unlike relational databases, graph databases are suitable for finding relationships within huge amounts of data without performing a join operation. Use cases for graph databases are where relationships among data is as important as data itself such as location based services, social networking websites, knowledge representation, recommendation systems, and path finding. The commonly used graph databases are Neo4j [18], Titan [19], AllegroGraph [20], InfiniteGraph [21], and InfoGrid [22].

Key-value databases organize data as an associative array of entries consisting of key-value pairs. Each key is unique and is used to retrieve the values associated with it. Lookup time in key-value databases are very efficient comparing to relational databases, which make them highly suitable for applications where schema is prone to evolution. Most key-value stores favor high scalability over consistency. Popular key-value stores are Riak [23], Voldemort [24], Redis [7], Dynamo [25], SILT [26], Memcached [6], HERD [3], FaRM [4], Nessie [27], Pilaf [2], HydraDB [1].

2.2 Remote Direct Memory Access (RDMA)

Cluster-based computing are becoming popular for a wide range of applications such as distributed deep learning [28–30], and distributed in-memory/disk data storage [31, 2, 4]. These systems are typically built from connected computers with high speed Local Area Networks (LANs). Much research has been carried out in order to improve the communication performance in cluster-based computing either by optimizing the existing protocol [32] or inventing new communication standards. A great deal of work on high-performance communication, such as Arsenic Gigabit Ethernet [33], U-Net [34], VIA [35], (Myricom/CSPi)'s Myrinet [36], Quadrics's QSNET [37] has led to modern high-speed networks including InfiniBand [38], RoCE [38], iWARP [39], and Intel's Omni-Path [40], which support Remote Direct Memory Access (RDMA) [41]. RDMA blurs the boundary of each machine by creating a virtual distributed, shared memory among connected nodes, i.e., substantially reducing communication and processing on the host machine.

RDMA allows direct memory access from the memory of one host to the memory of another one. Examples of such network fabrics are *internet Wide-Area RDMA Protocol* (*iWARP*) [39], *RDMA over Converged Ethernet* (*RoCE*) [38], and *InfiniBand* [38]. In contrast with conventional protocols, RDMA implements the entire transport logic in network interface card—commonly known as *Host Channel Adapter* (*HCA*)—to boost the performance.

Although iWARP, RoCE, and InfiniBand protocols provide a unique set of operations, however adopting an appropriate protocol requires the awareness of advantages and drawbacks of each protocol. The iWARP [39] is a complex protocol published by *Internet Engineering Task Force* (*IETF*) which only supports reliable connected transport [42]. It was designed to convert TCP to RDMA semantics which limits the efficiency of iWARP products. On the contrary, RoCE is an Ethernet-based RDMA solution published by *InfiniBand Trade Association* (ITA) supporting reliable and unreliable transports. InfiniBand is an advanced network protocol with low latency and high bandwidth commonly used in commodity servers.

The *OpenFabrics Alliance* (*OFA*) publishes and maintains user-level *Application Programming Interface* (*API*) for the RDMA protocols (i.e., iWARP, RoCE, and InfiniBand) [43] along with useful utilities, called *OpenFabrics Enterprise Distribution* (*OFED*).

As Figure 2.1 shows, RDMA allows an application to queue up a series of requests to be executed by HCA. Queues are created in pairs, called *Queue Pair* (*QP*), to send and receive operations. An application submits a *Work Queue Element* (*WQE*) on the appropriate queue. Then, channel adapter executes WQEs in the FIFO order on the queue. Each WQE can work with multiple scatter/gather entries to read multiple memory buffers and to send them as one stream and write them to multiple memory buffers. When the channel adapter completes a WQE, a *Completion Queue Element* (*CQE*) is enqueued on a *Completion Queue* (*CQ*). RDMA provides various work requests consisting of different operations as following:

- `SEND/RECV` sends/receives a message to/from a remote node
- *Fetch-And-Add* (`FAA`) atomically returns and increments the value of a virtual memory location in the remote machine
- *Compare-and-Swap* (`CAS`) atomically compares the value of a virtual memory address with a specified value and if they are equal, a new value will be stored at the specified address
- `READ/WRITE` reads/writes a data from/to a remote node exploiting the *Direct memory Access* (*DMA*) engine of the remote machine (i.e., bypassing the CPU and kernel)

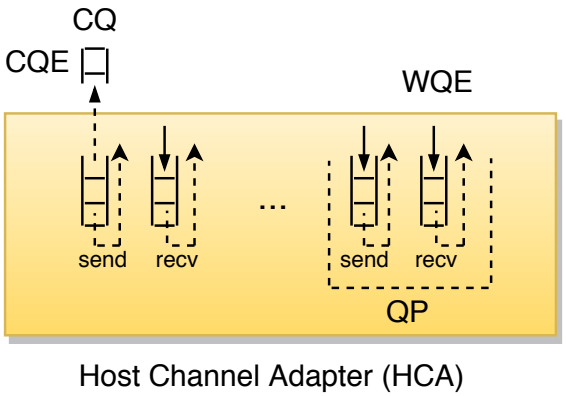

Fig. 2.1 InfiniBand Architecture.

Figure 2.2 shows a possible RDMA communication between machines A and B. Firstly, the user application issues a `SEND` request in step 1 to communicate with the channel adapter through CPU in step 3. Kernel space operation in the step

2 is used only for starting an RDMA connection, and there is not any operation or buffering when connection is set up. Furthermore, the existence of the fourth operation depends on the request type. If the data are inlined, the HCA does not need to perform an extra DMA operation to read the payload from the user-space memory. Afterwards, the request is enqueued in the send queue and is waiting for its turn to be processed by the *Network Processor* (*NP*). When the message is received by machine B, it can be performed on a memory address (step 4) without any reply to machine A or with a reply to machine A (step 5).

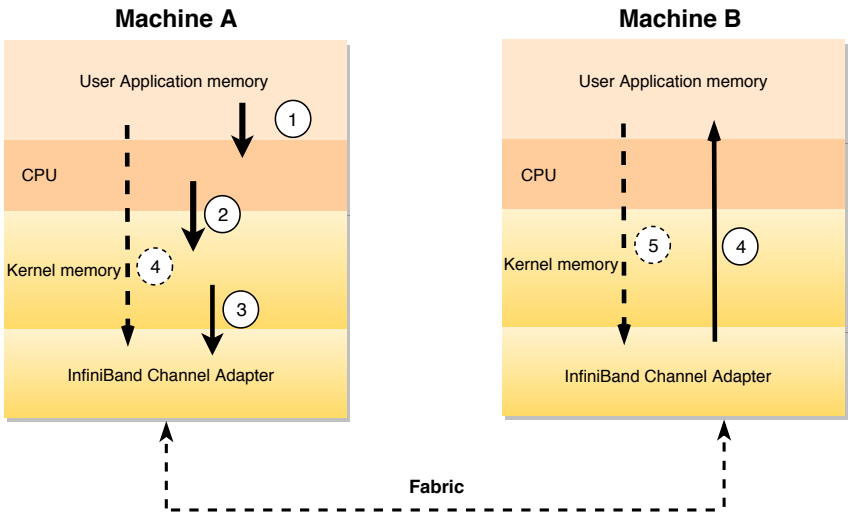

Fig. 2.2 RDMA communication cutting view.

2.2.1 Opportunities

RDMA-enabled protocols provide important features to accelerate data transfer without network software stack involvement (zero-copy), context switches (kernel bypass), and any intervention (e.g., cache pollution, CPU cycles) of the remote processor (no CPU involvement). These features allow RDMA applications to saturate the network using few cores, unlike traditional distributed systems. RDMA applications use CPU in an efficient way which can be beneficial in shared CPU environment (i.e., cloud solution). Furthermore, these features enrich RDMA-enabled application to achieve the highest communication throughput, in particular for small messages. RDMA provides a distributed shared memory over a reliable loss-less infrastructure through hardware-based re-transmission of lost packets [3].

Compatibility with legacy protocols is an important capability of RDMA-enabled protocols. For example, iWARP and RoCE are designed to be compatible with legacy Ethernet protocol, and InfiniBand supports legacy socket applications through *IP over InfiniBand* (*IPOIB*).

2.2.2 Challenges

Although RDMA provides higher performance compared with traditional protocols, the dilemma between RDMA performance achievement and redesign cost of an existing application according to RDMA semantic can be a challenging task. Redesign can be as broad as the whole system or limited to the communication component with the RDMA send/receive counterpart. It should be noted that the performance achievement by the latter approach is restricted.

One of the challenging problems in an RDMA-enabled distributed application is orchestrating local and remote memory accesses, since these are transparent to each other. Practically, synchronizing these concurrent accesses hinder the RDMA performance through the concurrency control mechanism. Typically, applying concurrency control incurs access amplification to guarantee the data consistency. However, there are some limited hardware-based solutions such as *Hardware Transactional Memory* (*HTM*). Furthermore, this concurrency problem is not limited to remote and local memory accesses, and there is a race among the remote memory accesses as well. For such condition, RDMA supports atomic operations such as `CAS` and `FAA`. The performance of these atomic operations intrinsically depends on its implementation in hardware [44]. Furthermore, the size of an atomic operation is limited (e.g., 64 bit). RDMA atomic operations can be configured to global (*IBV_ATOMIC_GLOB*) or local (*IBV_ATOMIC_HCA*) granularity. The local granularity guarantees that all the atomic operations to a specific address within the same HCA will be handled atomically, the global one guarantees that all the atomic operations on a specific address within the system (i.e., multiple HCAs) will be handled atomically. However, until now, only local mode is implemented in the existing HCAs.

2.2.3 Suitability

RDMA can perform one operation in each *Round Trip Time* (*RTT*), however traditional protocols provide more flexibility by fulfilling multiple operations. Thus, it makes RDMA more suitable for environments with single and quick operations. Furthermore, RDMA is more suitable for small and alike message sizes. Additionally, exploiting RDMA with dynamic connections cannot be beneficial due to the heavy initial cost of RDMA. Moreover, although RDMA provides high-speed communication, it limits scaling in both distance and size, meaning that the distance among nodes and the number of nodes cannot be arbitrarily large [45].

RDMA offers simple `READ`, `WRITE` and atomic operations (`CAS` and `FAA`) as well as an operation to send messages (`SEND`). However, it does not support sophisticated operations such as dereferencing, conditional read, on-the-fly operations, results consolidation [46]. Dereferencing indicates accessing to the content of a pointer to read its value. Conditional read represents reading memory based on a specific condition. On-the-fly operation means performing operations within the data transfer such as compression and decompression. Results consolidation indicates merging the result of multiple operations.

Chapter 3

Key-value store

A *distributed key-value store* offers a flexible data model with weaker consistency to partition data across many nodes on a computer network. Starting in mid-2000, numerous commercial key-value stores have emerged, each with its own unique characteristics, such as Google Bigtable [47], Amazon Dynamo [25], and Facebook Cassandra [12] to enable managing massively distributed data at unseen scale, which simply was not feasible with traditional relational database systems running on commodity hardware. These systems have become critical for large-scale applications, such as social networks [48, 25], realtime processing [49], and recommendation engines [50–52] to achieve higher performance.

Given the rise of key-value store over the last decade there have been two major efforts to accelerate NoSQL platform using modern hardware. The first approach was to employ storage-class memory, e.g., such as solid-state disks (SSDs), that focused on exploiting SSDs as a cache between main memory and disks [53], such as cassandraSSD [54], Flashstore [55], Flashcache [56], BufferHash [57]. The second approach has been to capitalize on the ever-increasing size of the main memory in each machine. These machines can now be connected through fast optical interfaces from a massive virtual shared memory space at an affordable cost such as RAMCloud [58], Memcached [6], MICA [59], SILT [26] and Redis [7].

Much research has been carried out in order to improve the communication performance either by optimizing the existing protocol [32] or inventing new communication standards.

A great deal of work on high-performance communication has led to modern high-speed networks including InfiniBand [38], RoCE [38], iWARP [39], and Intel's Omni-Path [40], which support Remote Direct Memory Access (RDMA) [41]. RDMA blurs the boundary of each machine by creating a virtual distributed shared memory among connected nodes, i.e., substantially reducing communication and processing on the host machine. Through RDMAs, clients can now directly access remote memory without the need to invoke the NoSQL's traditional client-server model. This motivates the NoSQL community to invest in developing purely in-memory key-value stores with RDMA capability, such as HydraDB [1], Herd [3], Pilaf [2], DrTM [5], FaRM [4]. RDMA capable protocol (i.e., InfiniBand) supports legacy socket applications through *IP over InfiniBand* (IPOIB); however, running existing in-memory systems on top of it can not efficiently exploit the benefits in the infrastructure [1, 2]. So existing in-memory key-value stores strive to reduce latency and achieve higher performance by exploiting RDMA operations [60]. In this chapter, commonly used underlying data structure and data concurrency in RDMA-enabled in-memory key-value store are discussed.

3.1 Hash Tables

The most common and efficient underlying data structure in modern in-memory key-value stores is based on hash table. Hash tables are dictionaries, where keys are mapped to a table with a hash function. Hash function is a one-way function which maps an input key to a value. The important properties of a hash table are lookup time, storage space, collision resolution, and hashing types.

The expected number of memory probes for all operations can be made arbitrarily close to 1 using a simple universal hash function by an appropriate load factor. However, worst case lookup time is important in particular in distributed hash table because the cost of each probe is particularly high. Therefore, the challenge is to hold constant lookup time with a reasonable space usage.

The critical point of every hash table is the handling of collisions. A *Collision* happens when different keys are mapped to the same location of the hash table by the hash function. Finding an alternate location for the collided key is called collision resolution. Typically, collisions are resolved by either *close addressing* or *open addressing*.

Close addressing (chaining) uses a link list through dynamic memory allocation to hold the key in the same index. However, open addressing resolves the collision by probing through a set of alternate locations in an array of buckets [61].

Hash function has a key role in distributing keys over the hash table. Mainly, there are two types of hashing: *static* and *dynamic* hashing. In static hashing, when a search-key value is provided, the hash function always computes the same address. The problem of this approach is that it does not expand or shrink dynamically as the size of the database grows or shrinks. Dynamic hashing provides a mechanism in which data buckets are added and removed dynamically. In short, "close" always refers to some sort of strict guarantee that objects are always stored directly within the hash table (closed hashing). Then, the opposite of "close" is "open" in which there is not such guarantees.

3.1.1 Chained hashing

Chained hashing scheme consist of an array of buckets each with a linked list of items, as can be seen in Fig. 3.1 [62]. Since this hashing mechanism has dynamic memory allocation, it implies dynamic memory reclamation and concurrency control. Furthermore, link traversal in this approach causes cache pollution which harms the performance [63].

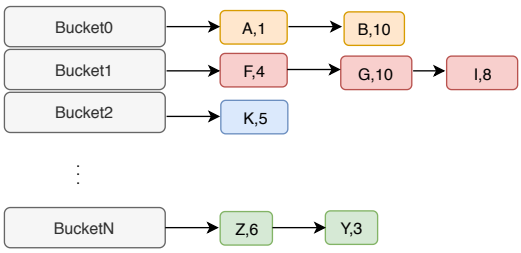

Fig. 3.1 Chained hashing.

3.1.2 Cuckoo hashing

Cuckoo hashing is a dynamic hashing which uses two hash tables and two hash functions. Every key is stored in one of the hash tables but never in both. The algorithm is named cuckoo similar to the birds nesting habits. Cuckoo hashing provides worst case constant lookup time and a good memory utilization. Experimental analysis

shows that cuckoo hashing is appropriate for modern computer architectures and distributed environments. The main idea in cuckoo hashing is using different hash functions and tables to resolves collisions. If the place presented by the first hash function is occupied it goes for the next location in the second table.

3.1.3 Hopscotch hashing

Hopscotch hashing is a static hashing scheme with the open addressing method for collision resolution in hash tables. Hopscotch hashing is interesting because it guarantees a small number of lookups to find entries. The main idea behind hopscotch hashing is that each bucket has a neighborhood of size H. The neighborhood of a bucket B is defined as the bucket itself and the (H-1) buckets following B contiguously in memory (H buckets total). This also means that at any bucket, multiple neighborhoods are overlapping (H to be exact). Hopscotch hashing guarantees that an entry will always be found in the neighborhood of its initial bucket. As a result, at most H consecutive lookups will be required in contiguous memory locations.

3.2 Concurrency control

Concurrency control deals with the issues involved with allowing simultaneous access to shared entities. Concurrency control is used to address conflicts with simultaneous access. Concurrency controls coordinates simultaneous transactions while preserving data integrity. In the following, the commonly used concurrency control in RDMA-enabled in-memory key-value stored are discussed.

3.2.1 Self verification

In self verification mechanism each hash table entry is protected by a checksum. Each entry stores a self-verifying pointer which contains a checksum covering the memory area being referenced. A self-verifying data structure allows clients to perform consistent reads in the face of concurrent writes. In this approach commonly a cyclic redundancy check (CRC)-n is used. CRCs are a type of error-detecting code used to implement checksums. CRCs are specifically designed to satisfy the property that they can detect transmission errors in data. The idea is to detect any data loss or

data corruption [2]. In order to find the appropriate n, the important points are the number of CRC execution in unit of time (N), and the collision probability $1/(2^n)$. Knowing them, the probability of a collision in unit of time is $(2^n)/N$. Through this computation the appropriate CRC-n can be chosen.

3.2.2 Versioning

Versioning relies on cache coherency of DMA [4]. Cache coherency guarantees the uniformity of shared resource data that stored in multiple local caches. When one bucket is stored in multiple cache lines, it stores version number of each bucket at the start of each consumed cache line. Through update, an object is updated by updating the data in each cache line. Through a read, the header version is checked with all the cache line versions. If the check succeeds, the read is strictly serializable with transactions. Otherwise, the RDMA is retried after a randomized backoff.

3.2.3 Flag and lease

For each key-value pair a guardian word is appended at the end of the value to indicate whether the data has been updated [1]. Upon receiving an update request, firstly the guardian word of the value is flipped in an atomic manner, then a new area for the updated key-value is created. In this approach, efficient memory reclamation is a challenging task, which is solved through a lease-based deferred memory reclamation to guarantees the data integrity. Lease is a time agreement which guarantee the memory availability in that period.

3.2.4 Hardware Transactional Memory

Since an Hardware Transactional Memory (HTM) provides strong atomicity and one-sided RDMA operations are cache-coherent [5]. Thus, combining these two can work as a concurrency control mechanism. The one-sided RDMA operation can access to the remote memory as a non-transactional operation, and HTM as a locall access on the target machine. In this case, any RDMA operation inside an HTM transaction will unconditionally cause an HTM abort and thus we cannot directly access remote memory through RDMA within HTM transactions.

Chapter 4

State of the art

Modern in-memory key-value stores [1], [3], [2], [5], [4] adopt RDMA to alleviate the communication and remote processing overheads. In this chapter, a comprehensive review of state-of-the-art RDMA-based key-value stores, examining indexing, consistency models, and communication protocols is provided. Furthermore, two legacy and well-known in-memory key-value stores are described in order to compare with RDMA-based systems.

4.1 HydraDB

HydraDB is a general-purpose in-memory key-value store designed for low latency and high availability environment [1]. HydraDB partitions data into different instances coupled with a single-threaded execution model, called *shard*. HydraDB adopts single-threaded model on an exclusive core to avoid shared memory access and context switching to fully exploit the computational power and the cache of each core. Each shard maintains a cache-friendly hash table with the location of the key-value stores instead of their actual content. This hash table (called *compact hash table*) is not visible to the clients and each bucket is aligned to 64 bytes with 7 slots and a header to an extended slot to avoid link list traversal. Each slot contains a signature of the key-value and a pointer. The server processes a request if its signature (i.e., a short hash key) matches to the requested key. Values are stored with a word-size flag in order to show the validity of the value content. Fig. 4.1 shows

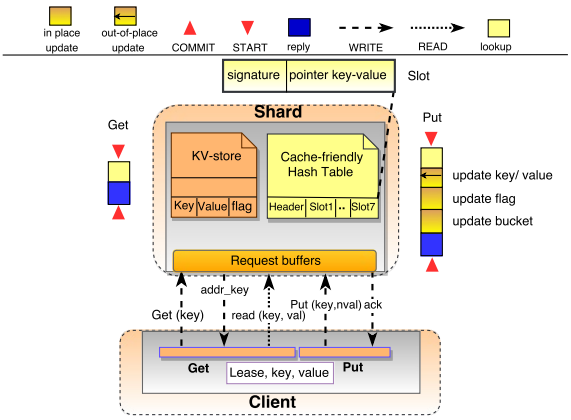

Fig. 4.1 Schematic view of HydraDB.

the schematic view of HydraDB. The scheme shows the indexing, communication protocol, and the required operations for *Get* and *Put* transactions.

HydraDB supports single-statement *Get* or *Put* transactions. Each client locates key-values according to the consistent hashing algorithm [64]. Clients and servers use `WRITE` to send and receive requests/responses, `WRITE` outperforms the other RDMA communications. In this method, the process keeps polling a memory area to detect a new arrival message (i.e., sustained-polling). Each shard uses a single thread to poll the buffer requests. In the case of *Get*, shard firstly finds the corresponding key-value address in the compact hash table. Then, it replies to the client the address of the key-value pairs. So for the next request of the same key from the client, it exploits the `READ` based on the cached address.

Put operation fully relies on the server; the server finds the key in the compact hash table, then it flips the flag of the key-value atomically to notify the readers about the update. Since a `READ` and a local write may conflict, shard exploits out-of-place update with lease-based time to guarantee the data consistency and memory reclamation, respectively [65].

4.2 Pilaf

Pilaf [2] is a distributed in-memory key-value store exploiting RDMA operation with the goal of reducing latency and improving the performance of traditional in-memory key-value stores (i.e., Redis [7] and Memcached [6]). As Fig. 4.2 demonstrates, Pilaf

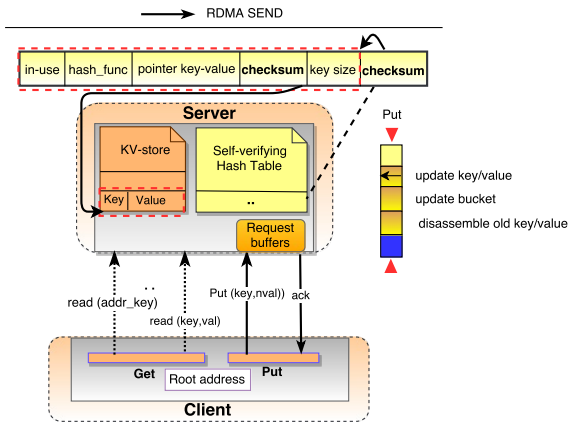

Fig. 4.2 Schematic view of Pilaf system.

exploits two distinct memory regions visible to the client: a variable extent area and a fixed-size self-verifying hash table. The extent area is the region to store the actual value. Each bucket in the fixed-size self-verifying hash table keeps the address of the key-value, its checksum, its value size, and the checksum of the bucket itself.

Pilaf supports single-statement transactions *Get* and *Put*. Unlike the *Get*, the *Put* operation is fully server-driven. The *Get* operation probes fixed-size self-verifying hash table through `READ` to find the appropriate bucket for the key with the valid content (i.e., *in_use*). Since the address of the value is stored in the bucket, client can read the value in the extent area. The *Put* operation fully relies on the server to avoid the write-write race condition.

The Pilaf client and server use `SEND` message to exchange request and response in *Put* operation. Once the Pilaf server receives a *Put* operation, first, it allocates a new memory location and updates it with the new value. Then it updates the corresponding bucket in the self-verifying hash table and disassembles the previous key-value content. Each bucket equipped with a checksum over the value to guarantee the data integrity. Disassembling the previous content notifies the clients about the recent update by the inconsistency between the read value and its checksum in the corresponding bucket. Moreover, each bucket is equipped with a checksum over the bucket itself to solve the race condition between the client's read and server's update on the same bucket. Once a client detects this inconsistency, it initiates the lookup in the hash table to retrieve the updated address of the key and its content.

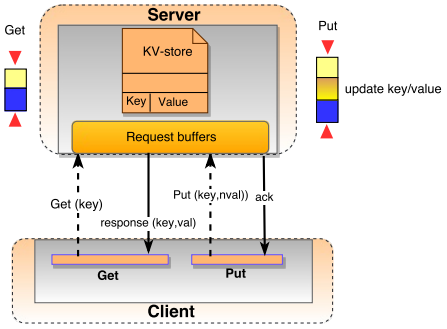

Fig. 4.3 Schematic view of HERD system.

4.3 HERD

HERD is an in-memory key-value store designed for efficient use of RDMA operations [3]. HERD adopts a simple lossy associative index, and a circular log for storing values (i.e., MICA back-end data structures [59]), illustrated in Fig. 4.3. The clients write their requests (i.e., *Get, Put*) to the server using `WRITE` on an unreliable connection and server replies using `SEND` over an unreliable datagram. Adopting these operations are due to the scaling of `WRITE` and `SEND` in inbound and outbound communications. HERD's designers claim that single RTT communications (i.e., `WRITE`, `SEND`) combining with a memory lookup can outperform multiple RTT communications (i.e., `READ`).

Although zero packet loss is detected in 100 trillion packets over unreliable datagram [66], there is the belief that unreliable transports may bring the unreliability of the enterprise applications [4]. HERD designers propose *FaSST* which is a transactional in-memory store with a loss detection algorithm over unreliable connection [66].

4.4 FaRM

FaRM is a distributed in-memory transaction processing system designed to improve the latency and throughput of the TCP/IP communication [4]. FaRM exploits symmetric model in which each machine uses its local memory to store data. This symmetric model helps to exploit the local memory and the CPU which is mostly idle. The FaRM adopts two memory areas for storing and handling transactions: a chained associative hopscotch hash table and a key-value store area. FaRM employs

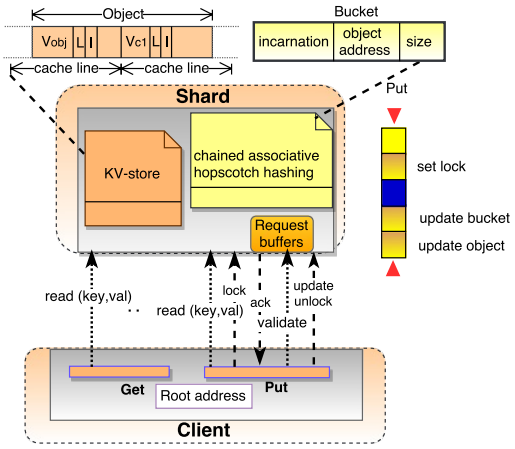

Fig. 4.4 Schematic view of FaRM system.

a modified version of hopscotch hashing [67] that uses a chain in the bucket. A chain is used to keep the new data in the bucket instead of resizing the table in the overflow situation. However, it attempts to remove this chain and to move the last element of the chain to the available slot. Each bucket in the hopscotch hash table consists of an *incarnation*, the address of the value, and its size. FaRM stores small value sizes into the bucket and the bigger sizes in the key-value store area and keeps its address in the bucket. Values in FaRM are stored in a structure called *object*. Each object consists of a header version (V_{obj}), a lock (L), an incarnation (I), and cache line versions (V_c). The incarnation is used to determine the validity in case of the removed object. Lock, header and cache line versions are used to guarantee the data consistency.

The FaRM supports multi-statement *Put* and *Get* transactions. Fig. 4.4 shows the FaRM data model and the interaction of the two transactions. *Get* operation uses `READ` for lookup process. It is performed by reading consecutive buckets according to the size of the neighbourhood (H) in hopscotch, where H=6 in FaRM. In addition, the client checks the lock and the header version to be matched with all the cache line versions to guarantee the data consistency. In case of failure, the client retries to read after a backoff time. In case of *Put* transaction, client fetches the desired key from the shard, then it locks the key by sending a request to the shard. Shard sets the lock atomically (i.e., compare and swap) and sends the acknowledgment to the client. Afterwards, client validates the key by reading the key and sends the update (i.e., key and updated value) to the shard. Shard updates the value and finally, the cache line and header versions are incremented and the object is unlocked. FaRM

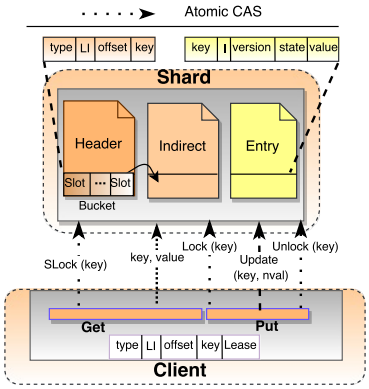

Fig. 4.5 Schematic view of DrTM.

uses fences after each memory write to guarantee the memory ordering in case of concurrent read. FaRM uses `WRITE` and sustained-polling mechanism to exchange requests and responses as HydraDB. However, this approach is incompatible with out-of-order packet delivery and retransmitted packet from an old message which might cause a memory overwritten and causes inconsistency in the execution [68].

4.5 DrTM

DrTM [5] and its successor DrTM+R [69] are in-memory key-value systems, which exploit concurrency instruction provided by modern CPUs. DrTM adopts traditional hash table with collocated memory regions for keys and values, called *cluster hashing*. Memory regions are managed in three different areas, called *main header*, *indirect header*, and *entry*, as shown in Fig. 4.5. The main header includes the incarnation, key, and its offset. The indirect header has the same structure as the main header and it is used in the overflow situation of the slots of the bucket in the main header. In this case, the last slot of the bucket points to an available indirect header. The value in DrTM is stored in a structure called *entry* containing the incarnation, value, version and status. Status represents the state of the key to perform *Get* and *Put*.

DrTM supports multi-statement transaction *Get* and *Put*. To guarantee the data consistency, it uses a lease-based in combination with lock and Hardware Transactional Memory (HTM). At the beginning of a transaction, the executor locks the remote key through the one-sided atomic RDMA verb (i.e., compare-and-swap) and fetches the keys, then a local HTM is started. DrTM uses the strong consistency

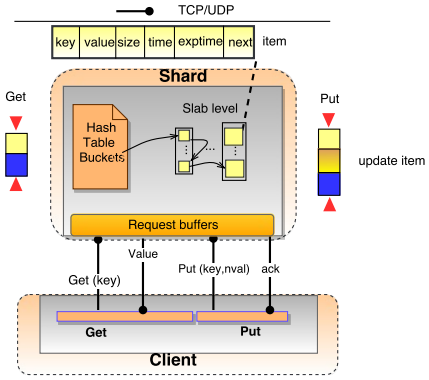

Fig. 4.6 Schematic view of Memcached.

and atomicity of RDMA and HTM, concurrent RDMA and HTM operations on the same memory will abort the HTM transaction. Once the transaction is committed, all remote keys are updated and locks are released.

4.6 Memcached

Memcached is a legacy in-memory key-value store based on TCP/IP protocol. It stores keys with their values into an internal hash table as shown in Fig. 4.6. It uses slab allocator to efficiently manage the memory according to the size of the key and value. Since Memcached supports multithreaded access to the hash table, the server orchestrates access through the locks. Memcached supports single-statement *Get* and *Put* transactions. Once the server receives a *Get* request, it finds the appropriate bucket and acquires the lock. Then, it replies the value or a miss, in case of missing the key, to the client. Once the server receives the *Put* request, it acquires the corresponding bucket lock, then updates the item. Memcached keeps an expiration (*exptime*) and the recent access time (*time*) to the key to replace the new items with the old ones in case of memory shortage according to the Least Recently Used (LRU) algorithm.

4.7 Redis

Redis is an in-memory key-value store with the ability to asynchronously store data on the disk. Flushing data to the disk can release memory space. Redis

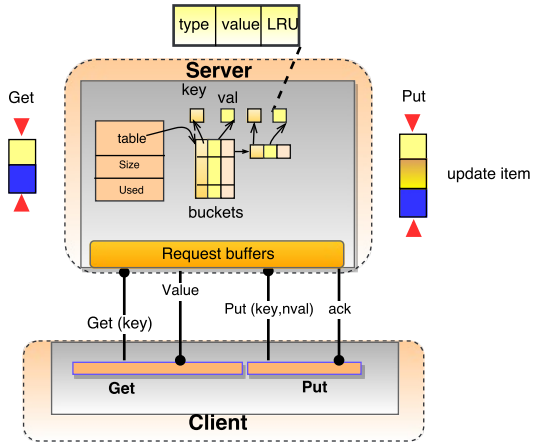

Fig. 4.7 Schematic view of Redis.

uses the TCP/IP communication scheme, and supports single-statement *Get* and *Put* transactions considering that it is single threaded and does not use locks for accessing the data. One of the main advantages of Redis is supporting the various value data types, such as lists and sets. Fig. 4.7 shows the communication and data model of the Redis. It uses *LRU* algorithm to flush the data to the disk; however, there is a high probability of data loss in case of system power loss events.

KV Store	One-sided / (Two-sided)	Request / (Response) connection type	Client-driven / (Server-driven) operations	Architecture model
HydraDB	✓/ (✗)	RC / (RC)	✗/ (Get,Put)	Asymmetric
Pilaf	✓/ (✓)	RC / (RC)	Get / (Put)	Asymmetric
HERD	✓/ (✓)	UC / (UD)	✗/ (Get,Put)	Asymmetric
FaRM	✓/ (✗)	RC / (RC)	Get / (Put)	Symmetric
DrTM	✓/ (✗)	RC / (RC)	Get,Put / (✗)	Symmetric
Memcached	✗/ (✗)	RC / (RC)	✗/ (Get,Put)	Asymmetric
Redis	✗/ (✗)	RC / (RC)	✗/ (Get,Put)	Asymmetric

Table 4.1 Decoupling the communication of the existing systems.

KV Store	Size Get / (Put) Amplification	Computation Get / (Put) Amplification
HydraDB	Word / (0)	lease validity + check flag / (0)
Pilaf	(0) / (0)	(2.6) × CRC64 / (0)
HERD	0 / (0)	0 / (0)
FaRM	V_{obj} + I + L + (#cache lines - 1) * V_{c1} / (0)	check lock / (0)
DrTM	2 × State + I + version / (2 × State)	0 / (0)
Memcached	0 / (0)	0 / (0)
Redis	0 / (0)	0 / (0)

Table 4.2 Decoupling the client amplification of the existing systems.

KV Store	Data Consistency	Indexing	Transaction
HydraDB	flag & lease	compact hash table	Single
Pilaf	self-verifying	cuckoo hashing	Single
HERD	✗	lossy associative index	Single
FaRM	versioning	chained hopscotch hashing	Multiple
DrTM	lock and HTM	cluster chaining hashing	Multiple
Memcached	lock	chaining hash table	Single
Redis	✗	chaining hash table	Single

Table 4.3 Data consistency and indexing of existing systems.

4.8 Systems Comparison

Table 4.1 differentiates aforementioned systems based on the usage of one-side or two-sided verbs, connection type in sending/receiving request/response, client-driven/server-driven operations, and architecture model of the systems. Client-driven/server-driven are operations that fully managed by the client/server. Architecture model captures the usage of client local memory in storing key-values.

Table 4.2 classifies the systems based on amplification in size and computation. Amplification highlights the extra amount of bytes or computation a client must exchange or compute in *Get* and *Put* operations. The word-size flag is the overhead of *Get* operation in HydraDB system. HydraDB client must check the flag and lease time which has computation overhead. In Pilaf, the client performs a lookup in the self-verifying hash table to find the address of the value, then it reads the value. To guarantee the consistency, the client must compute the checksum of the bucket and value. Client on average performs 1.6 probes in the self-verifying hash table plus one more `READ` to read the value. So on average, client requires computing 2.6 times the checksum. In FaRM, the header and cache line versions are amplification to the value. The DrTM requires to read and set the *State* in *Get* and *Put* operations. Moreover, *Get* operation requires to read an Incarnation (I) and version.

Table 4.3 shows the data consistency, indexing, and transaction type of the systems. In-memory key-value stores can be categorized to single-statement (also called *caching systems*) and multi-statement transactions. In the single-statement systems, such as HydraDB [1], Pilaf [2], Herd [3], Memcached [3], and Redis [7] there is one operation in the transaction. However, multi-statement transaction, such as FaRM [4], and DrTM [5], have multiple operations in the transaction.

Chapter 5

Performance challenges in modern systems

In high-performance applications not only board-to-board communication is critical, but also core-to-core, CPU-to-CPU, and I/O communications require careful investigation to explore the performance tradeoffs. For example, hardware message passing among cores [70–72], CPU-to-CPU Intel QuickPath Interconnect (QPI) and AMD HyperTransport (HT) [73, 74], and I/O PCI express, Intel Data Direct I/O [75, 76] all strive to enhance the communication performance.

Fig. 5.1 shows the schematic view of a node equipped with InfiniBand card [77–79] in a modern cluster. It shows the architecture of request queues in an InfiniBand channel adapter and the components of a system. Although the network architecture in the cluster affects the contention for resources, its impacts are outside the scope of this chapter. In this chapter, different components such as memory, host bus communication, HCA memory, RDMA features, network communication, and application level issues are investigated.

5.1 Memory

Although remote memory access through RDMA operation is quite fast compared to traditional network operations, they are still substantially slower than a local

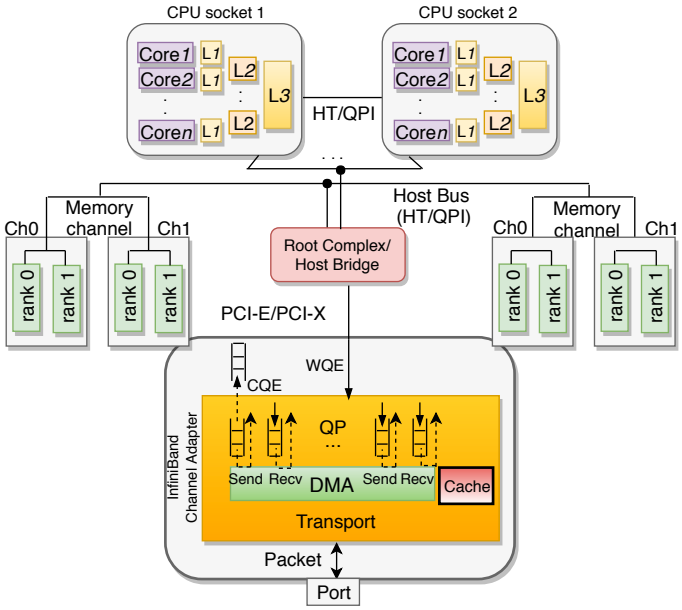

Fig. 5.1 System view.

memory access [4]. Thus, a better management of the local memory can influence the performance. The main RDMA challenges are described in the following.

Memory registration RDMA device requires to register a memory to read and write data from/to this memory. The cost of the memory registration can be divided into three parts: (1) mapping virtual to physical memory, (2) pinning the memory region (3) registering the memory region to NIC driver. Memory registration is a costly operation because of the kernel call and the write operation to the NIC driver. Pre-registering the memory can eliminate this cost in runtime. However, if the applications can not store their data in a pre-register memory, then data need to be copied within a register memory region.

[80] shows that comparing the cost of copying memory versus registering the new memory depends on the memory size and the power of the host. [80] shows that the cost of the memory registration can be even more than RDMA operation itself (i.e., `WRITE`). So different techniques are proposed to solve this problem. Registering a memory region resides the page of that region in the memory; if the page is not resident in the memory, then the cost of page fault is added to the mentioned 3 parts. So ensuring the residency of the memory page before registering a memory region can decrease the cost of the memory registration. Memory allocation from kernel space (i.e., *_get_free_pages*) and registering in the kernel (e.g., *ib_reg_phys_mr*)

instead of resorting to user space can decrease the memory registration latency [81]. Consequently, the first two steps of memory registration are eliminated since kernel memory space are physically contiguous and never swapped out. Since submitting a work request is not a blocking function, the overlapping memory registration with communication can hide the cost of registration. Yet comparing the cost of Round Trip Time (RTT) with the cost of memory registration reveals that it fully depends on the size of memory registration [80]. Parallel memory registration can also hide the cost. This technique is particularly effective when pages are resident in memory.

Cache miss The experiment on cache miss rate of the requester once the RDMA message uses different memory addresses (i.e., the cache is never hit), and once the message uses the same memory address (i.e., cache is always hit) shows that high cache miss rate of the requester can reduce the performance [82].

Data Alignment Since NICs work more efficiently on aligned data [83], using aligned message size can improve the performance of RDMA systems [82].

NUMA Affinity The distance between processor and data has a critical role on the performance. Generally speaking, better latency can be achieved through confining the memory access to local NUMA node. However, the appropriate deployment of processes and data can exploit memory bandwidth of other NUMA nodes [84–86]. Since the operating systems delegate the burden of NUMA-related issues on the application, designer must be aware of the data distribution on the main memory in order to reduce the latency and to increase bandwidth in memory access. [82] shows the impact of the NUMA affinity on the RDMA applications.

Memory Prefetching Software prefetching is a classical technique for overcoming the speed gap between the processor and the memory [87, 88]. Modern CPUs equipped with automated prefetching (e.g., Smart Prefetch) predict and preload the potentially needed data [89]. [3] shows that software prefetching in RDMA-based applications can improve the performance.

5.2 Host Bus communication

PCI Express (PCIe) technology [90] is an ubiquitous scalable, high-speed and serialized protocol standard for device communication and mainly a replacement for the PCI-X bus. Both PCIe and PCI-X allowed the device to initiate an independent

communication, called first-party DMA [91]. InfiniBand vendors nowadays adopt PCIe bus family for host communication due to the high-speed serial and dedicated link [92–94]. PCIe generations are evolved based on the speed of the link (i.e., lane speed and number of lanes), encoding, traffic, and packet overheads [95]. PCIe provides a root-tree based network topology, where all I/Os are connected, through switches and bridges, to a root complex. The root complex connects one or more processors and their associated memory subsystems.

Any movement through PCIe has an overhead on the performance, so it is important to understand the CPU and InfiniBand NIC interaction for high-performance applications. Aside from protocol and traffic overhead, *maximum payload size* and *maximum read request size* [95] may impact the performance in a PCIe system. These parameters might cause a limitation on transaction rate over PCIe. Though tuning these parameters have an impact on the performance of InfiniBand devices [96]. Moreover, interrupt request affinity on PCIe can improve application scalability and latency [97].

Profiling PCIe transactions are important to have a comprehensive view of the CPU-NIC interaction [44, 98]. Modern CPUs provide a list of events in Performance Monitoring Unit (PMU) to measure micro-architectural events of PCIe. For example, the events *PCIeRdCur* and *PCIeItoM* monitor DMA reads and writes from PCIe, respectively.

CPU can submit a work request to the NIC out of writing to the memory mapped I/O (MMIO) register (i.e., BlueFlame in Mellanox) or sending a list of works (i.e., Doorbell). It is recommended to use BlueFlame in the light load and Doorbell in high-bandwidth scenarios [99]. Each PCIe device equipped with a DMA engine can access to main memory independently. Firstly, the device sends a memory Read Request to the root complex. Then, it returns the desired memory by completion with a data packet. Comparing the CPU MMIO overhead with NIC DMA memory access reveals that the best trade-off is obtained by reducing the number of MMIOs [44].

The cache coherency between the NIC and CPU is a critical issue which is not written in the specification of InfiniBand protocol and is fully vendor specific. [4] reveals that there is a single cache line coherency for Mellanox adapters on *x86* processors. In addition, memory order on `READ`/`WRITE` verbs over PCIe are important concerns which are vendor specific.

Fig. 5.2 illustrates the execution path of `READ`, `WRITE`, `SEND`, and `FAA/CAS` operations. Firstly, CPU initiates operation by sending a work request to the HCA via *Mapped device memory* (*MMIO*) over PCIe. Then, NPP of HCA processes the WQE as following:

- In case of `READ`, the request are sent over fabric. Then, NPP of HCA in remote side handles the request by a DMA read over PCIe, which requires two PCIe transaction i.e, request data from memory (*MRd*) and read completions (*CplD*) to read the data. Afterwards, the requested data are sent back to the sender. HCA of the sender issues a DMA write over PCIe (*MWr*) to store the data in a pre-defined memory address.
- In case of `WRITE`, the HCA needs to fetch the payload by a DMA read. However, if the payload is inlined in the WQE this DMA operation is eliminated from the path. Next, the request is sent over fabric. HCA in the remote side fulfills the operation by a DMA write to store the data in a pre-defined memory address.
- In case of `SEND`, the HCA needs to fetch the payload by a DMA read as well as `WRITE` operation. Next, the request is sent over fabric. HCA in remote host requires to consume a `RECV` operation to determine the store memory address. Thus, it needs a MMIO action. Next, the payload is written by a DMA write.
- In case of `FAA/CAS`, the request are sent over fabric. Then, NPP of HCA in remote side handles the request either by an internal locking mechanism for the target address and issuing a read-modify-write or an atomic operation over PCIe. It should be noted that all intermediate routing elements of PCIe must support the atomic operation capabilities. Afterwards, the return value are sent back to the sender. Then, HCA of the sender issues a DMA write over PCIe (*MWr*) to store the data in a pre-defined memory.

Fig. 5.3 shows an analytical comparison of DMA operations over PCIe 3.0 according to the model presented in [100, 44]. A DMA read always uses less PCIe bandwidth than an equal-sized MMIO, and higher than a DMA write since there is not a response for a write. However, it should be considered that supported optimizations by the hardware can alter the presented performance [100].

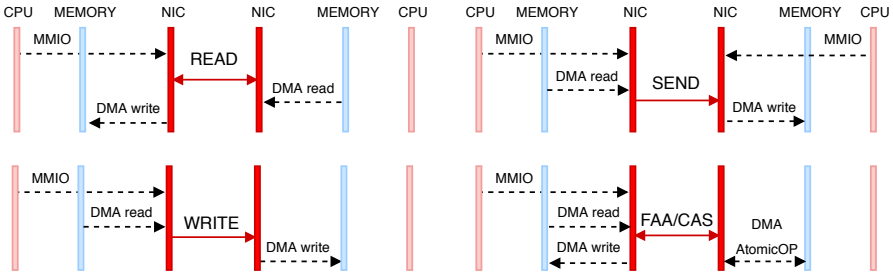

Fig. 5.2 RDMA operations execution paths.

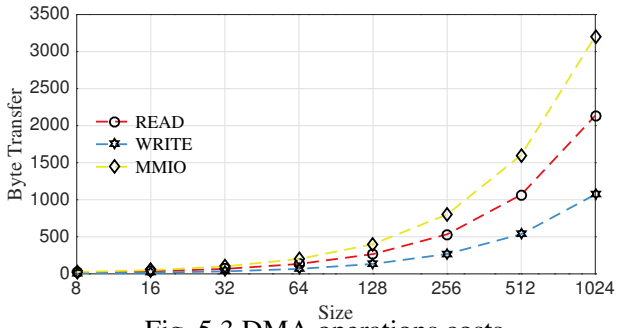

Fig. 5.3 DMA operations costs.

Considering the mentioned analysis, RDMA has asymmetric performance, thus it does not incur the same overhead at either sides. Since the destination memory address is carried by `READ`, `WRITE`, `FAA`, and `CAS` operations, they can fulfill without the intervention of the remote CPU. On the contrary, `SEND` operation doesn't carry memory information, and destination needs to post a receive WQE to capture the data. Thus, a synchronous rate between sending `SEND`s and `RECV`s is required. This rate can be managed either by *Shared Receive Queue* (SRQ) [79] or over provisioning the `RECV` in combination with a backoff between `SEND`s.

Roughly speaking, the overhead of `READ` is higher than `WRITE`, and `SEND` higher than `READ`. Furthermore, the performance of atomic `FAA` and `CAS` operations depends on the amount of parallelism in the workload with respect to the HCA's internal locking mechanism.

5.3 NIC Memory

First InfiniBand products (developed by Mellanox) provided memory on the NIC board. However, recently essential resources have been moved to the host memory

and only a cache memory remains on the board since the memory access time of the host does not have a significant impact on the performance [101].

The NIC cache memory, particularly in Mellanox adapters, are served for several purposes such as maintaining page tables of registered memory (to translate virtual to physical address) or queue pair (QP) data (i.e., state and elements) [44]. Since adapter has limited resources, the optimization of this scope can improve the performance. Adopting larger memory pages will reduce the number of entries of page tables, reducing fetching page table entries from system memory to NIC (i.e., page faults) [4, 80]. Reducing the number of queue pairs can reduce the memory usage [4]. In addition, the work request submission rate is quite important for avoiding cache miss in the NIC [44].

5.4 RDMA Features

Choosing the right RDMA features is critical to the scalability and the reliability of the application. In this section, several RDMA features and their impact on the performance are described in more detail.

Transport Type RDMA supports unreliable and reliable types of connections. Reliable connection guarantees the delivery and the order of the packet by the acknowledgment from the responder. An unreliable connection does not guarantee the delivery and the order of the message.

RDMA provides two types of transports: unconnected and connected. Each QP is connected to exactly one QP of a remote node in connected mode unlike unconnected (datagram). Since for each connected connection between two nodes, two QPs are required one for the requester and one for the responder, the number of QPs increases $2\times$ with the number of connections. There are different approaches to reduce the number of QPs. In a reliable connection, threads can share QPs to reduce the QP memory footprint [4]. Sharing QP reduces CPU efficiency because threads contend for the cache lines for QP buffers between CPU cores [66, 102]. In the Annex A14 of the InfiniBand specification 1.2, eXtended Reliable Connection (XRC) was introduced to connect nodes [103]. Multiple connections of a process in a node can be reduced to one.

Shared receive queue (SRQ) shares receive queue on multiple connections and reduce the number of QPs. SRQ solves the two-sided communication synchronization problem between the requester and the responder which previously was solved by using backoff in the requester and over provisioning of receive WQEs in responder [79]. SRQ can solve this problem since an incoming receive message on any QP associated with an SRQ can use the next available WQE to receive the incoming data.

Inline Data Inlining the data to the work queue element (WQE) eliminates the overhead of memory access through DMA for payload after submitting a WQE and expecting the performance raising. However, an inline message has limitation according to the size of the payload.

Message Size The size of the message could be bottlenecked in two places: host bus communication (i.e., PCIe) or the Path Maximum Transfer Unit (PMTU). *Maximum payload size* and *maximum read request size* in the PCIe communication affects the performance of memory access from the InfiniBand adapter [95, 96]. It basically specifies the number of essential completion with data packets. The higher read-request size increases the efficiency of packet transfers. When a QP (reliable/unreliable connected) is created, the PMTU is determined in the queue and if the desired message to be sent is larger than the PMTU of the queue, the message is divided into multiple messages. However, if InfiniBand receives a message larger than its port Maximum Transfer Unit (MTU) it silently drops the message [104].

Reducing the number of cache lines used by a WQE can improve throughput drastically [44]. Roughly speaking, increasing the size of the message increases the communication latency. [44] demonstrates that increasing the size of the message will decrease the performance.

Completion Detection While InfiniBand adapter completes a work request, it enqueues a CQE in the completion queue. Mainly, two approaches can be adopted to detect completion of a work request: busy polling and event handling. In the former mechanism, the application polls the completion queue to receive a CQE. This approach has high CPU utilization; however, the cost of polling is quite low since the operating system is bypassed. In the second approach, a notification is received when a CQE arrives to the completion queue. This approach is much better based on CPU utilization. However, it requires the operating system intervention.

Busy polling outperforms event handling in all possible RDMA operations [82, 105]. However, in large message size, both methods converge.

Completion Signaling A work request can be sent with signaled or unsignaled opcode. If the opcode of the work request is set to signaled, once the work request is completed a work completion element is generated. Unsignaled opcode generates no element to the completion queue and consequently, there isn't extra overhead. However, the latter approach cannot be adopted due to the resource depletion, and a signaled work request must be sent periodically to release the taken resources by unsignaled work requests. Finding the best period to send a signaled work request is a challenging task. The send queue depth and the message size can be considered as parameters in finding the best period [82].

Batching Once the CPU sends a list of requests to NIC instead of sending one request per each is called *batching*. The advantage of batching is reducing the number of CPU-NIC and network communications due to coalescing the requests. However, hardware limitation does not allow to batch requests of different QPs [44]. The batching scheme is more appropriate for the datagram transport due to its intuitive multicast support. So this scheme can be used to batch requests over datagram connections to multiple remote QPs. In addition, sending multiple requests in a message is another approach that allows the requester to send several requests to a specific responder and amortize communication overheads [48].

Atomic operations RDMA intrinsically provides a shared memory region in a distributed environment. Cross-access to the same memory region must be handled in order to avoid the race condition. RDMA supports two types of primitives to avoid concurrent access from other RDMA operations (not only atomic) on the same NIC: *fetch-and-add* and *compare-and-swap*. These operations adopt an internal lock mechanism of the NIC. The performance of these primitives depends on both the NIC atomic implementation mechanisms and the level of parallelism [44]. The atomicity of RDMA CAS is hardware-specific with different granularity (i.e., IBV_ATOMIC_HCA, IBV_ATOMIC_GLOB) [106]. However, there is no concurrency control between the operation from local and the remote RDMA operations. To solve the concurrency problem, concurrent data structures [2, 4] have been proposed or special hardware instructions providing strong atomicity (i.e., hardware transactional memory in Intel processor) are exploited [5].

RDMA support protocols InfiniBand is different from Ethernet in different aspects [107]. There are several InfiniBand alternative protocols supporting the RDMA technologies including RoCE and iWARP. Adopting an appropriate protocol requires the awareness of their advantages and drawbacks. The iWARP [39] is a complex protocol published by Internet Engineering Task Force (IETF). It only supports reliable connected transport (i.e., it does not support multicast) over a generic non-lossless network [42]. It was designed to convert TCP to RDMA semantics. On the contrary, RoCE is an Ethernet-based RDMA solution published by InfiniBand Trade Association supporting reliable and unreliable transports over lossless network [68]. There are several studies comparing these protocols over time [108–111]. The consensus is that iWARP is unable to achieve the same performance of InfiniBand and RoCE.

Wire Speed Mellanox InfiniBand has been made in 5 speeds: Single-Data Rate (SDR), Dual-Data Rate (DDR), Quad-Data Rate (QDR), Fourteen-Data Rate (FDR), and Enhanced Data Rate (EDR) offering 2.5, 5, 10, 14, 25 Gbps respectively, so enhancing the link speed increases the performance and decreases the latency [112].

Adaptor Connection InfiniBand adapters are based on PCI-X, PCIe. Different studies show that NIC based on PCIe outperforms PCI-X [93, 113, 114]. InfiniBand host communication has been started to be integrated on the chip of the processor [115, 116, 44].

5.5 Communication

RDMA allows several communication paradigms based on its primitives. There are two main actors on each communication, those that make requests (clients) and those that respond to the requests (servers). Communication can be categorized as synchronous or asynchronous according to the type of message passing. In a synchronous communication, the client sends a message, and it is blocked until a response arrives, then it returns to its normal execution. In this mode, a message represents a synchronization point between the two processes. In an asynchronous communication, the client sends a message, then continues its execution till the response is ready. Since in an RDMA-based synchronous communication there is not a common clock between client and server to agree on a data transfer speed,

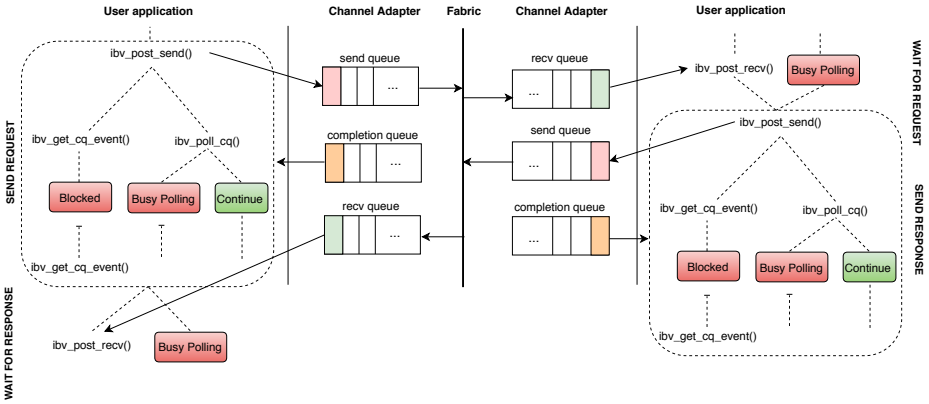

Fig. 5.4 Synchronous and asynchronous RDMA communications.

busy polling or blocking functions is exploited. Meanwhile, an RDMA-based asynchronous communication can be implemented through events with notification.

Fig. 5.4 shows possible synchronous and asynchronous RDMA communications. Firstly, the client sends its request by posting a work request to the send queue through `ibv_post_send()`. Then, it receives the completion of its work request through `ibv_get_cq_event()` or `ibv_poll_cq()`. `ibv_get_cq_event()` is a blocking function while `ibv_poll_cq()` cannot make block with any flag or timeout parameter, thus it can be blocked by busy polling. According to the type of communication, the client can either use `ibv_post_recv()` or busy polling to get its response. Generally speaking, the server receives the request and responds to it on the same manner but in the opposite order.

When statements in a transaction require to be executed sequentially, a synchronous communication is exploited. Synchronous communications are challenging due to the blocking essence of them. This blocking mechanism incurs a bottleneck in a system. Therefore, more attention is required in order to select a proper synchronous communication paradigm to achieve the best performance. Thus, this section focuses on synchronous implementation of the main communication paradigms. However, the possibility of asynchronous implementation of each paradigm is discussed as well.

Fig. 5.5 shows the main communication paradigms without middleware layer exploiting the above-mentioned communication primitives. In model (a), the client and the server exchange a pre-defined memory address for requests and replies. The server is polling the memory address for a new request, and the client writes its

request to the pre-defined memory address in the server. Then, the server replies by writing to the client memory address. In model (b), the client writes its request in a local memory address and the server polls this memory to find a new request. Afterwards, the server replies to the client by writing to a pre-defined memory in the client. In model (c), the server is almost passive and the client writes its request in a pre-defined memory address in the server and polls to check the response. In model (d), the client writes its request in a local memory address, and the server polls over this memory to find a new request. Then, the client polls in a pre-defined memory in the server to check the response. In model (e), the client sends a request message to the server, then the server returns a response message to the client. Finally, model (f) is the traditional socket communication request and reply.

Model (a), (b), and (c) rely on sustained-polling mechanism for synchronous communication, which can be implemented in different manners [1, 4]. Typically, polling is relying on the last packet of a written message to detect request completion. However, the correctness of this polling-based approach depends on the delivery of the message in order to avoid anticipated last packet delivery or memory overwritten by an older message [68].

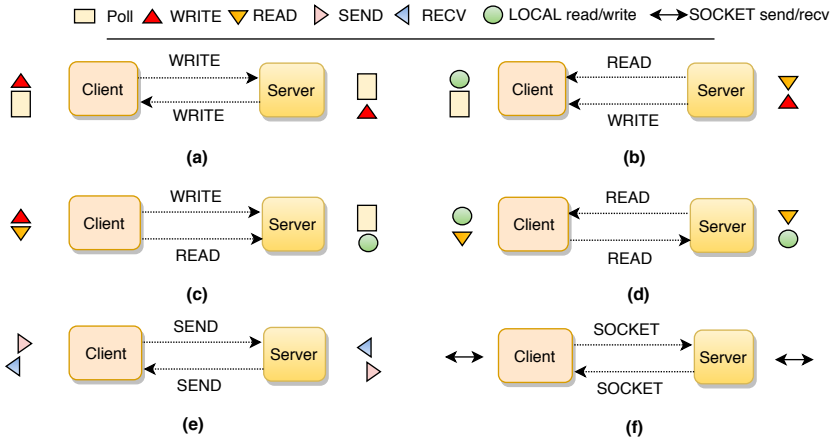

Fig. 5.5 Communication paradigms.

According to the mentioned definition, model (a) can be fully implemented in an asynchronous manner through `WRITE` with immediate data, and synchronously by the polling mechanism. Although the client in model (b) can be implemented asynchronously, the server can not be asynchronous due to the lack of notification in `READ`. Model (d) doesn't support asynchronous communication in either sides. Model (e) and (f) can be implemented in both manners.

METHOD	CLIENT OVERHEAD	SERVER OVERHEAD	NETWORK TRAFFIC	COMMUNICATION	CONNECTION
WRITE-WRITE	high	high	low	a/synchronous	RC/UC
READ-WRITE	low	high	high	synchronous	RC
WRITE-READ	high	low	high	synchronous	RC
READ-READ	fair	fair	high	synchronous	RC
SEND-SEND	high	high	low	a/synchronous	RC/UC
SOCKET-SOCKET	high	high	low	a/synchronous	RC/UC

Table 5.1 Comparing communication paradigms.

As can be seen, each paradigm has its own unique characteristic which makes it suitable for a particular environment. Model (a) and (d) share the effort for communication in a fair manner. Model (c) puts the burden of communication on the client side and model (b) on the server side. Model (a) incurs higher CPU usage in both client and server sides due to polling. Model (b), (c), and (d) generate more network traffic for polling remote side. Model (e) and (f) don't induce a pre-defined memory address but they require synchronization in sending and receiving messages.

Each communication paradigm supports a series of connection types according to the adopted RDMA operation. RDMA supports unreliable and reliable connection types. Unlike unreliable, reliable connection guarantees the delivery and the order of the packet through an acknowledgment message from the receiver. Furthermore, RDMA supports unconnected and connected connections. Unlike unconnected, each QP in connected communication connects exactly to one QP of a remote node. This chapter only considers the connected connections, i.e, *Reliable Connected* (*RC*) and *Unreliable Connected* (*UC*). Table 5.1 compares different communication paradigms according to client and server overheads, network traffic, communication, and connection types. Each model is named according to the operations presented in Fig. 5.5.

5.6 Application level issues

Application level parameters that impact on the performance are presented in the following.

Data Structures Hash table is a popular data structure in a multi-client and multi-threaded server environment due to its fast direct lookups; however, hash collision is inevitable, which leads the increased number of probes. The higher number of probes in a hash table naturally increases the cost of lookup and pollutes

the CPU cache by keys which are irrelevant. In many modern RDMA-based NoSQL systems [4, 2], Cuckoo and Hopscotch hashing are often employed [63, 67]. These hash tables strive to have the constant lookup cost by keeping the key in a bounded neighborhood to its original hash position.

Pipelining Pipeline allows simultaneous tasks at different stages. [117] proposes pipeline for memory registration and communication in order to hide the cost of registration. This approach can improve the performance depending on the size of memory. However, [1] compares multi-threaded request handling pipeline versus single threaded request handling which multi-threaded request handling harms the performance.

Flow Isolation Latency-sensitive and throughput-sensitive applications may need to share the network resources (i.e., NIC) in large-scale environment. The application deployment is critical in such scenario to avoid the performance isolation of either types of applications. [118] shows that in presence of both type of applications, a latency-sensitive flow will suffer. So throughput-sensitive and latency-sensitive flows are better to be isolated.

In-bound vs. Out-bound Requests can be categorized to *inbound* and *outbound* according to the requester [3]. Sending the request from multiple clients to one server is called inbound, and sending requests from one server to multiple clients is called outbound. Before designing an RDMA-based application, measuring inbound and outbound throughput is important. Outbound is bottlenecked by PCIe and NIC processing power while inbound is bottlenecked by NIC processing power and InfiniBand bandwidth [44].

Chapter 6

Kanzi: RDMA-enabled in-memory key-value store

After a comprehensive analysis of the state of the art RDMA-enabled in-memory key-value store and challenging performance issues in modern RDMA based clusters, Kanzi was proposed. Kanzi distributes data across a set of servers, where each server is further partitioned into a set of shards (referred to as Kanzi shard). Each Kanzi shard consists of 3 important parts: memory groomer, structure manager, and memory pool. Memory pool is a logical storage of each shard. Structure manager handles the mentioned structures in the memory pool. Memory groomer is operated by an exclusive thread to reclaim unused memory in memory pool. Fig. 6.1 shows the architecture of the Kanzi consisting of Kanzi shards and Kanzi clients.

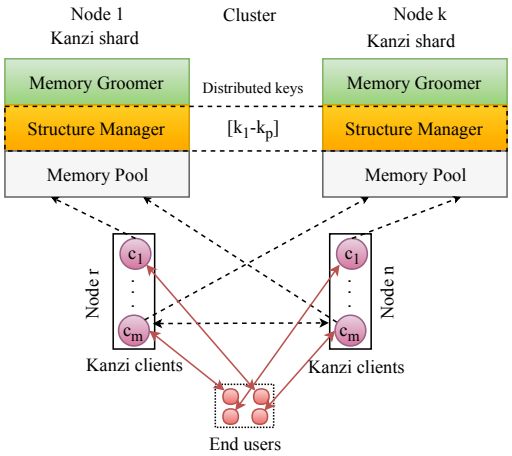

Fig. 6.1 Kanzi Architecture.

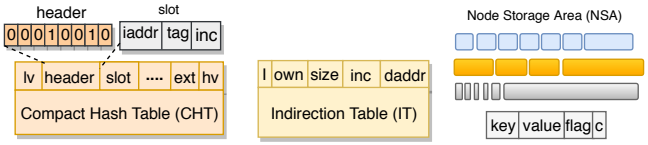

Fig. 6.2 Kanzi shard data structures.

6.1 Structure

As can be seen in Fig. 6.2, each shard exposes three logical memory regions: *Compact Hash Table* (*CHT*), *Indirection Table* (*IT*), and *Node Storage Area* (*NSA*).

Each bucket in compact hash table is 64 bytes which can be located in one cache line. It consists of a *header*, 5 slots and a pointer (*ext*) to extend the bucket when there is not an empty slot. Header indicates the free/busy status of each slot. Furthermore, each bucket includes two fields low version (*lv*) and high version (*hv*) for the sake of data consistency. Each slot includes 4 bytes indirection address (*iaddr*), 2 bytes tag (*tag*), and 1 bit incarnation (*inc*). The *iaddr* indicates key offset in the indirection table, and the *inc* bit determines the validity of the key in the system. The compact hash table is only updated by server to insert a new key-value item in the system. Clients only use this table at membership time to construct their local indirection hash table. Multislot and cache-line size of bucket diminish the membership time through avoiding a long linked list traversal and lower cache misses.

Indirection mechanism already has been employed to enhance the performance of traditional relational databases [119–121]. It is a pointer to the latest version of a record in the system. According to our experiment, maintaining an indirection pointer inside a key-value structure incurs a link-list, which hinders the performance due to long list traversal. Thus, the use of an indirection table was proposed. Clients in our design access to the latest version of each key-value item through the indirection table. Each entry in the indirection table consists of immutable field (*l*), ownership (*own*), *size*, *inc*, and the address of the associated key in the storage area (*daddr*). The l field is used for data consistency to avoid write-write race, the own filed determines the node that the corresponding key resides, size indicates the size of value, and inc represent the validity of data.

Node storage area keeps the actual key-value items in the system. In the design, a head pointer in NSA points to a free memory area and memory behind the head pointer is already allocated. A memory groomer in the background always keeps

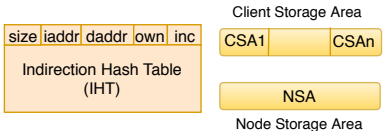

Fig. 6.3 Kanzi client data structures.

track of the allocated memory. When it finds an unused memory chunk, it reclaims it. This design was chosen due to the fast memory allocation and reclamation which is interesting for high-performance dynamic memory management. Each key-value item in node storage area consists of *key*, *value*, *flag*, and commit (*c*) fields.

As can be seen in Fig. 6.3, clients on a node has a dedicated and a shared memory region. Each client has its own storage area called *Client Storage Area* (*CSA*), and there is a shared region called *Node Storage Area* (*NSA*). NSA is a particular hybrid memory region since it is shared for all clients resident on the node, and also accessible from remote clients. Each node has a shared Indirection Hash Table (*IHT*) which keeps the essential information to access each key. Each bucket in IHT consists of *size*, *iaddr*, *daddr*, *own*, and *inc*. *iaddr* is the offset of the corresponding key in the IT, *daddr* indicates the address of the key-value item in the NSA of the owner. Finally, *inc* is the incarnation of the corresponding key from the CHT. An important point in client data structure is the size of the IHT and NSA. We exploit over-provisioning memory size in IHT to avoid any loss. However, in case of limited memory a lossy mechanism can be replaced to evict one key from the memory to purge IHT for new key. Moreover, invalid memories in NSA are reclaimed with a background process.

6.2 Kanzi Protocol

Kanzi's protocol is supported by RDMA investigations in Section 7. It basically provides four single statement transactions: *GET*, *PUT*, *INSERT*, and *DELETE*. End users in the system connect to any Kanzi clients through the conventional TCP/IP protocol to request their keys. When a Kanzi client receives an end user's request, then a set of messages are exchanged between Kanzi clients and shard on behalf of the end user to perform the request from the appropriate shard. Kanzi clients employ a mesh-based topology that the memory of all machines in the cluster are exposed as a shared address space. A single QP on each node is responsible for managing the

distributed memory region. This mechanism reduces significantly the maintaining QP list on each node instead of connecting each client together.

In order to avoid the overhead of concurrency control and to efficiently utilize the CPU, each Kanzi shard is managed by a single thread and is mapped to a single core which exclusively manages a partition of keys. Partition-based design may not be a suitable choice to handle highly skewed workloads [1], however exploiting the *IHT* to cache the latest updated items can alleviate skewed workload contention. Since the location of the data and its movement across NUMA nodes in Kanzi shard can directly influence the access latency, the memory allocation and access within the same memory node are confined.

Lazy synchronization A Kanzi client at the membership process reads the entire CHT of the shard and constructs its own IHT. Since the CHT is shared among all the clients and shard, a read-write race can happen. A non-blocking mechanism called *lazy synchronization* was proposed which allows concurrent access to the CHT.

Each bucket in CHT is enclosed with two fields: *lv* and *hv*. A bucket is updated exclusively by the responsible shard firstly by updating the *lv* field, then by writing the slots, and finally updating the *hv*. The IB specification does not guarantee cache line atomicity or write ordering, however the cache-coherency of DMA of NIC (on a x86-based system) exhibits each cache line according to the memory barriers between writes [4]. Thus, writes are separated by compiler memory barriers to provide the essential ordering. Furthermore, according to the best of our knowledge the READ reads the memory in left-to-right order (it is completely vendor specific feature). Thus, the writing from right-to-left (opposite direction) guarantees the consistency. Once a client reads a bucket, it checks the *hv* and *lv* fields. The equivalence of these fields guarantees the consistency of the read data.

GET transaction Kanzi provides a latch-free *GET* transaction to be more adoptable in read-intensive environment. Each *GET* starts firstly by a lookup in the IHT followed by a `READ` in NSA of the owner node to access the actual key-value item. Afterwards, the client must verify the validity of the item by checking the flag entity. If the flag is set, the client must access IT on the shard to read the latest update. If the item still exists (controlling the *inc*), it again performs a READ to access the actual key-value item. One of the main advantages of the mentioned mechanism is the constant lookup time for either small and large key value items. Pilaf [2] optimistically requires two `READ`s (one cuckoo hash table, and

one storage area) to fulfill a *GET* transaction. FaRM [4] fixes the lookup time in chained hopscotch hashing table at the expense of higher payload size to read a complete neighborhood of a bucket which incurs overhead. However, Kanzi fixes the lookup time regardless of the key-value item size through decoupling the index and actual values, and proposing the indirection table. Such light implementation of *GET*, puts the burden of keeping consistency on *PUT* transaction. The challenges of coping with consistency is described in the *PUT* section.

PUT transaction Although decoupling design enhances flexibility with respect to the location of stored data, it incurs overhead on *PUT* transaction to update both index and actual value in case of out-of-place update. Furthermore, it requires the invalidation of the previous version to keep consistency. Thus, one of the challenging problem in *PUT* transaction is compromising between data consistency and performance due to multiple points of update. Consistency control must handle clashes among *PUT*s and *GET*s such a way that avoid hindering the performance. Thus, Kanzi came up with a light concurrency mechanism.

When a Kanzi client receives a *PUT* transaction, firstly it stores the new key-value item in its local NSA with uncommitted state (*C*), then performs a lookup in IHT to find the desired entry. Next, Kanzi client updates the corresponding entry in IT. This update basically locks the entry by setting the *C* field, and passes the ownership (*own*) of the key-value item to the node that client resides and set the key-value address (*daddr*). The new uncommitted version is in-stalled state till fulfilling the transaction and committing the new version through updating *C*. Since client operations are transparent, shard can not perform any concurrency control. So, *PUT* transaction exploits atomic operation (CAS) to wipe any possible inconsistencies. Atomic CAS guarantees the data consistency if there are in-flight operations on the same entry at the time of memory update. This approach is feasible since each entry in the IT is managed in 64 bits which can be engaged by RDMA atomic operations. RDMA atomic operations can be configured to global (*IBV_ATOMIC_GLOB*) or local (*IBV_ATOMIC_HCA*) granularity. The local granularity guarantees that all atomic operations to a specific address within the same NIC will be handled atomically, the global one guarantees that all atomic operations on a specific address within the system (i.e. multiple HCAs) will be handled atomically. However, until now, only local mode is implemented in the existing NICs. Thus, in our deployment all the clients send their request through a single HCA, and each HCA handles request of one particular IT to avoid inconsistency. After updating the IT entry,

PUT transaction invalidates the previous version (*flag*), then it unlocks the IT entry and resets the commit (*C*) field in the local NSA. Fig. 6.4 illustrates the possible states of a key-value item with its transition. A new uncommitted key-value item has *valid:uncommit* state which will be changed to *valid:commit* once the *PUT* transaction is committed. Within the *PUT* transaction Kanzi client invalidates the previous version of item which causes *valid:commit* to *invalid:commit* transition. It should be noted that there is not *invalid:uncommit* state to avoid a concurrent update. Once the memory is reclaimed it changes to *reclaimed* state to be used for new items.

After transferring the ownership of a key-value item, the new owner must handle all requests for the migrated item but the corresponding shard still retains the associated entry in its indirection table. Such mechanism allows us to use the local memory of the clients to store data. Moreover, subsequent requests from the same node can be handled from the shared memory NSA and eliminates the network communication overhead. However, increasing the number of updates can incur performance degradation due to the data migration. The data migration can be beneficial for large size values to save the network bandwidth. However, the symmetric model and data migration can raise the performance if updated keys are hosted on the same node. Otherwise, it incurs a burst network traffic due to data migration among nodes. To inhibit this phenomenon, clients must be partitioned and connected to a particular shard.

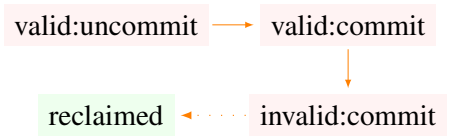

Fig. 6.4 State transition.

INSERT & DELETE transactions Kanzi clients are not allowed to insert a new key-value item. However, they are enabled to delete an existing one. When a client receives a *DELETE* transaction, the key-value item is evicted by incrementing the *inc* field in the corresponding entry in the IT. It performs the *DELETE* transaction by an atomic CAS to the corresponding entry in the IT. All the insert operations are performed through the Kanzi shard by allocating a new item in the CH and the IT. This policy enables the Kanzi shard to provide the logging facility to asynchronously logs all *INSERT* operation to the local disk similar to the other key-value stores such as Redis [7].

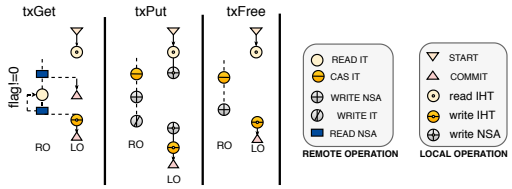

Fig. 6.5 Transactions.

Kanzi programming model seeks to let users work at a much higher level of abstraction than dealing with the internal components. We implemented Kanzi in C using the *libibverbs* library from the OpenFabrics Alliance. Kanzi provides a set of interfaces like `txGet()`, `txPut()`, and `txFree()` to support *GET*, *PUT*, and *DELETE* transactions respectively. Fig. 6.5 shows the internal mechanism of each interface.

Chapter 7

Experimental Evaluation

Since our study firstly focused on performance challenges of RDMA operations, it allows us to capture bottlenecks of RDMA-based in-memory key-value stores. Thus, a comprehensive set of unified experiments are designed and performed under the same conditions for the sake of fair comparison in order to evaluate the state-of-the-art systems. In this approach, the bottleneck of different approaches can be captured through a deep analysis.

Redis 3.2.9 [7], Memcached 1.4.37 [6], and HERD [122] are employed from the original source. However, HydraDB, Pilaf, and FaRM are implemented from scratch since they are not publically available. All messages in HydraDB and FaRM are exchanged using the inline RDMA messages. Since DrTM uses a special limited CPU feature that is not widely accessible, it is not included in the analysis. FaRM is the only system that supports multi-statement transaction, which is unified to be comparable to the other systems. The only modification is in the *Put* operation in which client sends the key and new value to the shard for update request as it presented in [3]. In addition, Redis and Memcached are executed over IPOIB and the other systems have native RDMA over InfiniBand support.

7.1 Settings and operation noise

To provide the reproducibility and interpretability of the experiments, the non-deterministic and deterministic parameters which can impact on the performance are described in the following.

Non-deterministic Setting Query distribution (i.e., object popularity) is one of the main parameters in the experiments. Facebook analysis reported that web requests follow Zipfian-like distribution [123] [124] and the majority of in-memory systems present experiments based on the Zipfian distribution with high α ratio ($\alpha = 0.99$) indicating skew curve [4, 1, 3, 2]. Zipfian distribution refers more generally to frequency distributions of rank data which can indicate the contention in the request. For comprehensiveness, two distributions are considered that closely model the real-world traffic: *Uniform* and *Zipfian*. Uniform distribution represents that each interval of the same length are equally probable.

The number of keys is an important parameter in the experiment. For example, the server needs to register corresponding memory size to the number of keys which can increase the cache misses in the NIC to fetch the page table entries and influence on the performance. In the experiment, 4 million key-value pairs with 16 and 32 bytes sizes were considered close to real-world workloads [48, 125].

Real-world workload ratio of read and write vary from *68%* to *99%* [126]. However, various read-write ratio workload ranging from read-only to write-only are considered.

Deterministic Setting All benchmarks are compiled with the gcc version 4.4.7 with 50 seconds warmup and 25 seconds measuring time. In addition, to achieve the certainty on the result the experiments are repeated three times.

Benchmarks are executed on a machine with 2 sockets AMD Opteron 6276 (Bulldozer) 2.3 GHz equipped with 20 Gbps DDR ConnectX Mellanox on PCI-E 2.0 with offload processing. The network topology is a direct connection with a Infiniscale-III Mellanox switch. Each machine has 4 NUMA nodes connecting to two sockets.

Each process is mapped to a single core to avoid context switching and uses its local NUMA node. Since NIC (i.e., Mellanox adapters) over PCIe is closer to one of the CPUs on the board [96], an experiment is conducted to examine the impact when both sender and receiver are mapped on the closer CPU and once on the far CPU to NIC on two machines. Fig. 7.1 shows the bandwidth difference on `SEND` operation when the processes are bind to close and far CPU from the NIC. According to the result, performance is higher when processes are pined to the closer CPU to the NIC.

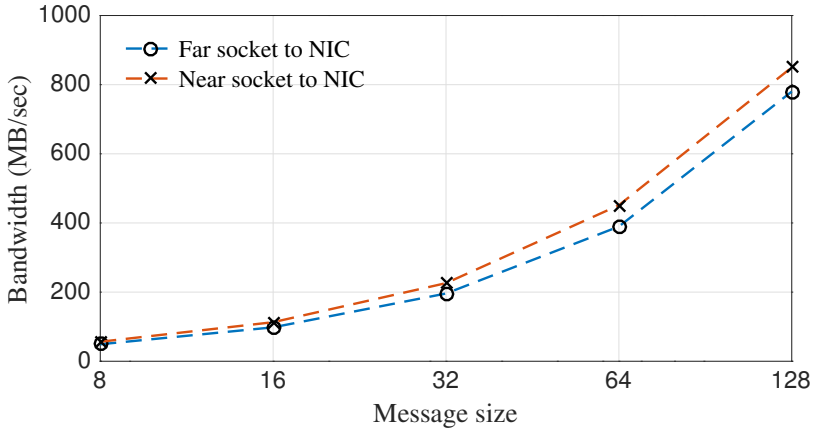

Fig. 7.1 RDMA Send bandwidth difference on far and close socket to NIC.

To measure the variation in the throughput of RDMA operations, four client processes reside on one host connected to one server process on another host, performing consecutive RDMA operations. This experiment has been repeated 100 times and each time a client is executed for 50 seconds. The throughput of each client have been measured, then they are unified to find the total value. Figure 7.2 shows the results of this experiment. As can be seen, the variance between the observed throughputs is negligible, with only a few outliers.

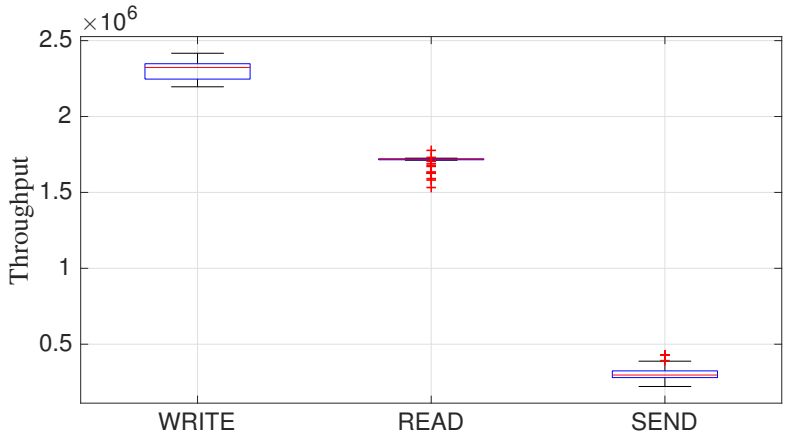

Fig. 7.2 Variation on the RDMA operations.

7.2 Optimization Experiments

Optimizations make a substantial difference in the overall performance of RDMA-based applications. Thus, primarily the experimental results of applying optimizations on RDMA operations are evaluated.

Payload size - Fig. 7.3 shows the impact of payload size on the throughput. In the experiment, one client performs RDMA operations with different payload sizes. As can be seen, increasing the payload size incurs the performance degradation, since the payload size affects the CPU-HCA interactions as well as the number of exchanged messages. Moreover, Fig. 7.3 illustrates that `WRITE` delivers an higher throughput than `READ` despite both operations have identical InfiniBand path. The reason behind is that `WRITE` requires less states to be maintained both at the RDMA and at the PCIe level [3]. In a `WRITE` operation, the client does not need to wait for a response. However, a `READ` request must be maintained in the client's memory till a response arrives. Furthermore, at the PCIe level, `READ` is performed using heavier transactions comparing to `WRITE`. In addition, `SEND` presents much lower performance comparing to the `READ` and `WRITE`, since it does not bypass the remote CPU, and it requires `RECV` at the server side which is a slow operation due to its DMA interaction for writing data and CQE [3].

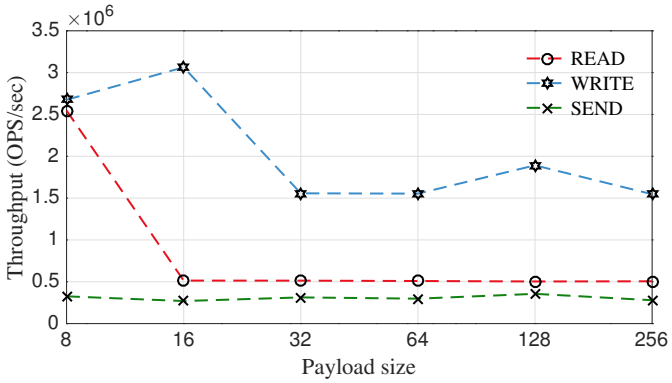

Fig. 7.3 Payload size impact on performance.

Unsignaled operation - When an WQE completes its operation, it pushes a completion signal to the CQ via a DMA write. Pushing this signal adds extra overhead on the operation due to its PCIe interaction [3]. Fig. 7.4 shows the throughput of RDMA operations when a selective signal is used. In this experiment, consecutive operations with 8 bytes payload size are sent unsignaled, i.e., completion

signals are not generated for these operations, then a signaled operation is sent to release the taken resources. It should be noted that the number of unsignaled operations cannot exceed the size of the queue pairs, which is set to 2048 units in this experiment. As Fig. 7.4 illustrates, increasing the unsignaled operations increases the performance.

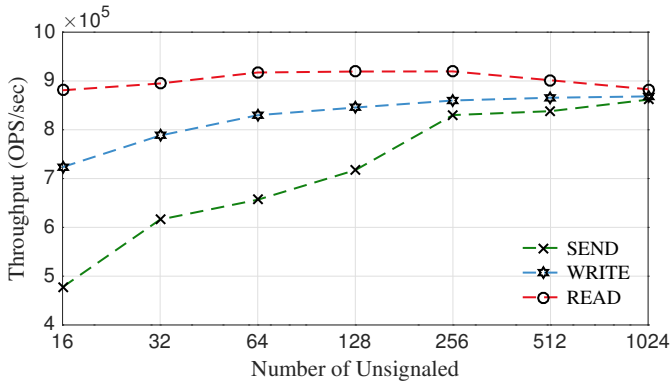

Fig. 7.4 Unsignaled operations impact on performance.

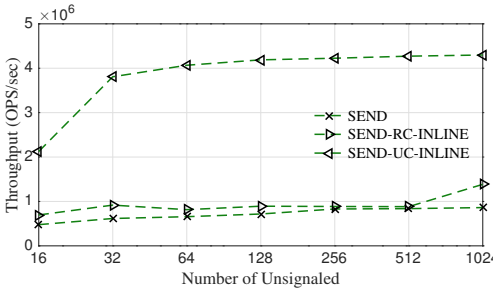

Fig. 7.5 Inline and connection type impact on the `SEND` performance.

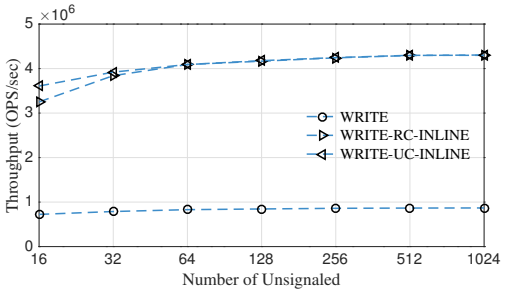

Fig. 7.6 Inline and connection type impact on the `WRITE` performance.

Inline message - An WQE can inline its payload up to the maximum programmed input/output size, otherwise the payload can be fetched via a DMA read [3]. Increasing the WQE size will increase the MMIO and DMA operations in the inline and non-inline messages. The key difference between the inline and non-inline RDMA operations relies on the fact that CPU puts the payload in the message, and non-inline message requires to read the payload by a DMA read operation [44]. Although inline message has higher throughput, it poses a limit on the message payload size. For example, it must be less than 1K bytes in the NIC employed in

our evaluation. Moreover, increasing the payload size causes higher throughput degradation in inline message comparing to non-inline messages.

Connection types - Although reliable and unreliable connections (RC/UC) have the same header size (i.e., 36 B), they present different performances. For example, both connected transports (RC/UC) require as many queue pairs as the number of connections in HCA comparing to unconnected connections. These queue pairs can increase memory consumption in HCA and may consequently affect the performance because of the increased number of cache misses.

Fig. 7.5, 7.6 show the impact of inline messages and connection types on `SEND` and `WRITE` operations while increasing the number of unsignaled operations. `READ` operation is not reported since it only supports reliable connections. Furthermore, it does not transfer payload, thus it does not support inline features. In this experiment, one client performs RDMA operations with 8 bytes payload size. As can be seen, the impact of inline and connection type is not the same on `SEND` and `WRITE` operations. `WRITE` is more sensible to inline and `SEND` is more sensible to connection type. These features can increase the performance up to 4.9 times in either cases.

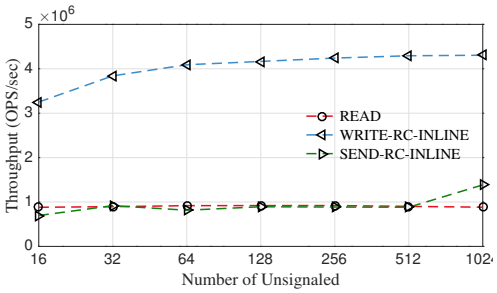

Fig. 7.7 Performance comparison on unreliable connections.

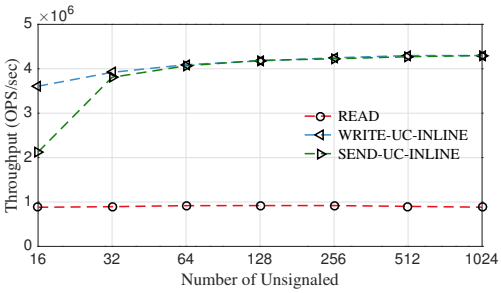

Fig. 7.8 Performance comparison on reliable connections.

Fig. 7.7, 7.8 illustrate the comparison of `READ` with `SEND` and `WRITE` in reliable and unreliable connections. As can be seen, `WRITE` with inline payload outperforms the other RDMA operations in both RC and UC connections.

Scaling is an important experiment because in real cases more than one client is typically connected to the server. So, an experiment was devised to investigate the impact of scaling on the performance. In this experiment, several clients send their requests towards the server. The server processes new requests in a round robin

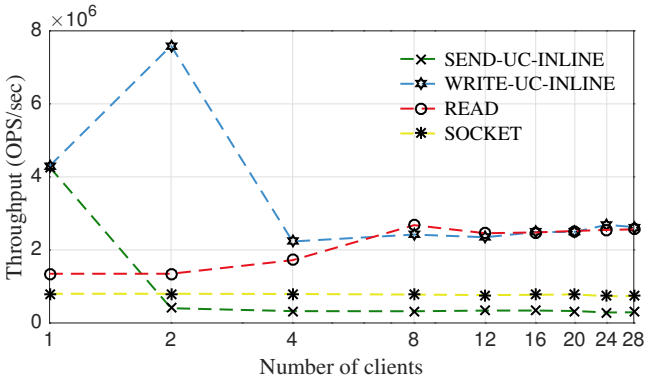

Fig. 7.9 Scaling different operations.

fashion. If the server finds a request from a client, it responds to the request. Each client is mapped to one dedicated core to avoid context switching. Each client sends 8 bytes message request with 512 unsignaled operations.

Fig. 7.9 shows the performance by increasing the number of clients. The degradation of `SEND` and `WRITE` throughput occurs because the `SEND`s and `WRITE`s are unsignaled, i.e., client processes get no indication of operation completion. This leads the client processes overwhelming with too many outstanding operations, causing cache misses inside the HCA. As can be seen, `WRITE` and `READ` outperform the other operations. Furthermore, with few clients the `WRITE` outperforms the `READ` operation. Surprisingly, `SEND` underperforms socket communication with higher number of clients. It should be noted that socket is implemented in non-blocking mode in this experiment. Furthermore, only one machine is adopted to deploy the client processes. However, increasing the number of client machines can alter the performance because the overhead of maintaining the QP states is distributed among different machines [44].

Fig. 7.10 Throughput of communication paradigms.

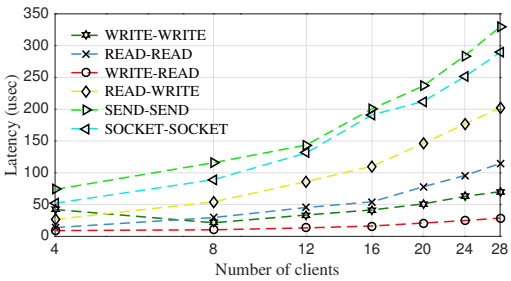

Fig. 7.11 Latency of communication paradigms.

Communication paradigms - Fig. 7.10, 7.11 show the performance and the latency of communication paradigms, respectively. According to the experiments reported in Fig. 7.9, `WRITE` operation has the best performance, thus it was expected that *WRITE-WRITE* outperforms the other communication paradigms. Surprisingly, the *WRITE-READ* has much larger (i.e., 2.4 $\times$) throughput than *WRITE-WRITE*. The reason is that in *WRITE-WRITE* communication, a client sends its request and polls the response area, and increasing the number of clients overwhelms the server to process requests and replies to them by `WRITE` operation. However, *WRITE-READ* paradigm lightens the server burden, since server replies to requests by writing to local memory. So, it can process more requests and reduce the response time of clients, consequently achieving a better throughput. Furthermore, Fig. 7.11 demonstrates that the latency of *WRITE-READ* paradigm is the lowest among the other paradigms. Unlike *WRITE-READ*, *READ-WRITE* overloads server by reading requests through `READ` and replies by `WRITE`. So, the *READ-WRITE* underperforms the *WRITE-READ*. The *READ-READ* has better throughput than the *READ-WRITE* because server only performs a `READ` operation comparing to a `READ` and a `WRITE` in *READ-WRITE* paradigm. As demonstrated in Fig. 7.10, *SEND-SEND* and *SOCKET-SOCKET* perform poorly due to their heavy operations, which was expected based on the results in Fig. 7.9.

WRITE-READ is the only paradigm in the experiment that scales well by increasing the number of clients, however the primary drawback of this approach is that the client requires more effort in exchanging request-response. Thus, it can not be a good option for environments where clients are placed with other processes on the same machine because the amount of CPU that must be allocated to client processes is significantly high. *READ-READ* is a good option to act fairly in both client and

server sides. However, it performs 3.9 times lower than *WRITE-READ*. Moreover, *READ-READ* incurs huge network traffic to poll remote host.

7.3 Analyzing state of the art

To evaluate the state-of-the-art RDMA-enabled in-memory key-value stores a set of unified experiment are designed to measure throughput, latency, fairness, and uniformity ratio. These metrics are chosen since they are commonly used metrics in highly cited papers in this scope. In order to stress each system, one machine is dedicated as the server with a single shard and one machine with multiple clients in the experiments.

7.3.1 Throughput

Impact of varying the number of clients and workload read-write ratio on throughput Fig. 7.12 shows the results on different workload read-write ratios on an Uniform distribution. As expected, the throughput of Redis and Memcached is an order of magnitude less than RDMA-based system in particular in a read-intensive environment. HydraDB scales well by increasing the number of clients and almost outperforms the other systems. In read-intensive environment, we explain this by considering the smaller amplification in the HydraDB and the use of `READ` for *Get* statement.

HydraDB reads only the desired value in case of valid lease time. FaRM uses hopscotch hashing which requires reading a neighborhood consisting of 6 buckets; thus, FaRM results in a heavier lookup.

Although Pilaf uses `READ` in *Get* statement, it can not outperform HydraDB and FaRM due to the higher number of `READ`s and cost of CRC64. In particular, the latency of CRC64 computation can even increase higher than `READ` latency [65].

HERD uses `WRITE` over unreliable connection for sending request and `SEND` over datagram for its response. Each client creates one QP to send its requests while the server creates one QP for each client to receive the requests. The server uses one QP for all clients to respond their requests and each client creates one QP for each server to receive the response. This mechanism puts more overhead on the clients due to

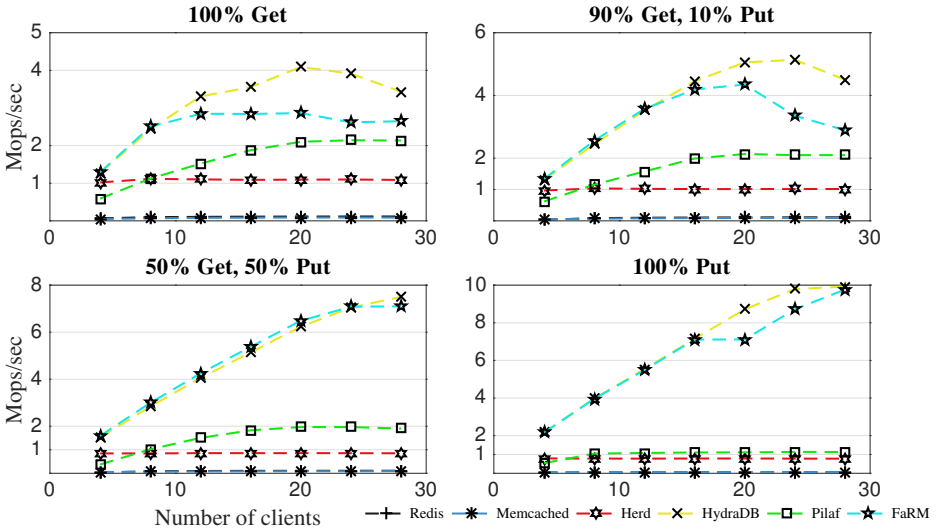

Fig. 7.12 Uniform throughput with single shard.

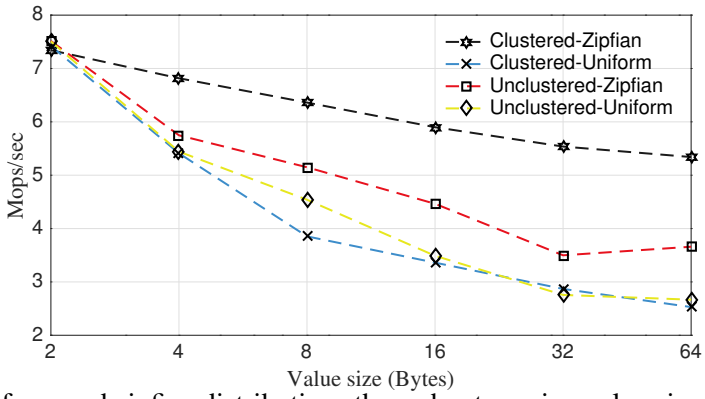

Fig. 7.13 Uniform and zipfian distributions throughput varying value size for clustered and unclustered insertions.

the higher number of QPs. In the experiment, HERD is bottlenecked with the `SEND` operation [3]. Moreover, HERD is executed without the hugepages which have an impact on the performance.

Throughput impact of the memory access distribution To isolate the impact of the memory access distribution on the performance, an experiment has been performed with 24 processes reading remote memory either using Zipfian and Uniform distributions. We deploy data in the clustered (i.e., in order) and unclustered (i.e., random) way. We observed that both distributions perform identically with the small value size due to the locality of the data in the smaller portion of the memory.

However, increasing the value size results reducing the locality and Zipfian memory access outperforms (i.e., 2.1×) the Uniform due to higher data locality, as can be seen in Fig. 7.13. Furthermore, Zipfian distribution with clustered deployment has higher (i.e., 1.4×) performance comparing to the unclustered deployment due to the higher data locality. Generally speaking, Zipfian distribution increases the chances of reusing cached data for hotkeys. However, it will increase the race condition when updating hotkeys. Fig. 7.14 shows the throughput results for Zipfian distribution.

Throughput impact varying the workload read-write ratio To highlight the impact of decreasing the workload read-write ratio for both Zipfian and Uniform distributions, an experiment has been devised with 24 clients while varying workload read-write ratio. As can be seen in Fig. 7.15, HydraDB and FaRM scale gracefully when decreasing the read-write ratio due to the higher performance of `WRITE` exploited in *Put* transaction comparing to the `READ` in *Get* transaction. The performance difference of `READ` and `WRITE` is due to the limited number of outstanding `READ` requests for each QP (i.e., our case 24) and the overhead of maintaining their state in the NIC. Moreover, `READ` uses PCIe non-posted transaction comparing to cheaper posted transactions for `WRITE` [3]. A non-posted transaction is a type of PCIe request, which requester needs a response from the destination device unlike the posted transaction. Pilaf does not scale with decreasing the read-write ratio due to increased overhead of two-sided verb in *Put* operations. In the case of HERD,

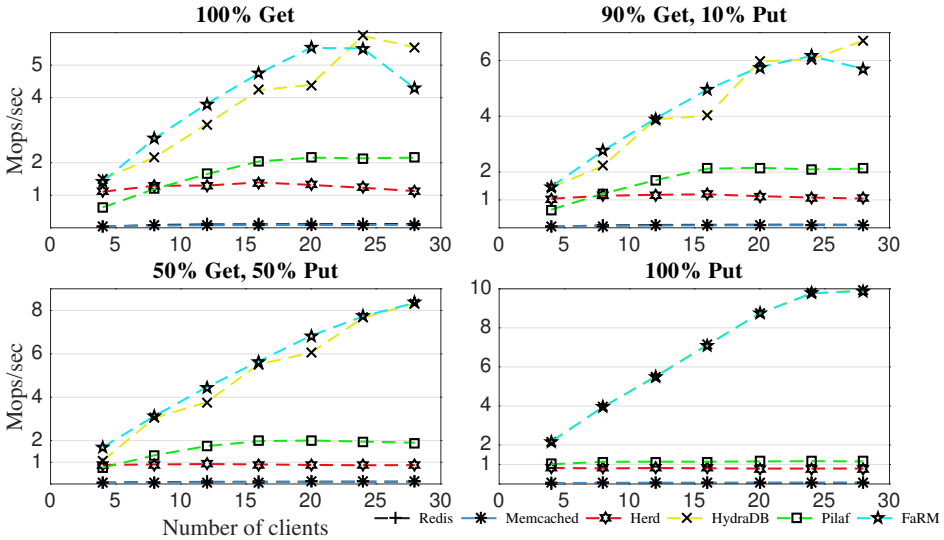

Fig. 7.14 Zipfian throughput with single shard.

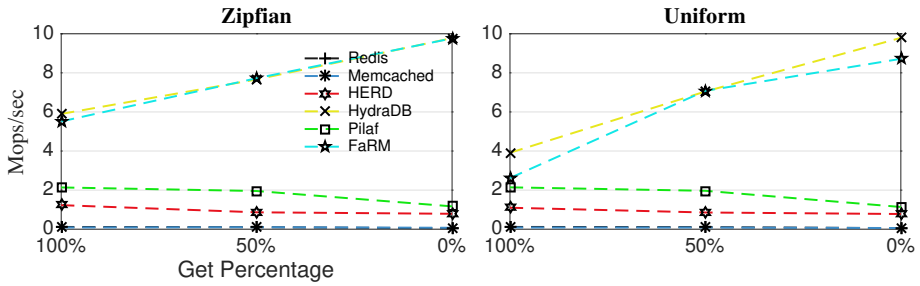

Fig. 7.15 Throughput comparison when varying read-write ratio for 24 clients.

decreasing the read-write ratio does not have a noticeable impact on the performance, since HERD is bottlenecked by the two-sided verb.

7.3.2 Latency

Fig. 7.16 and 7.17 show the latency of Uniform and Zipfian distributions. Increasing the number of clients will increase the number of requests and consequently will increase the latency.

HydraDB and FaRM on both distributions are two order of magnitude lower than legacy systems and decrease 2.5 and 2.3 times by decreasing the read-write ratio respectively. The drop is due to the lower latency of `WRITE` operation compared to `READ` used in *Put* and *Get* operations. However, decreasing the read-write ratio will increase the latency up to 2 times on Pilaf due to the higher latency of the verb messages comparing to `READ`. Since HERD uses the same operations in both *Get* and *Put* operations, there is no noticeable impact on the latency by decreasing the read-write ratio. Moreover, the latency of the Redis and Memcached are higher than RDMA-based systems because the use of the IP over InfiniBand instead of RDMA operations.

7.3.3 Value size

The payload size of RDMA message influences the throughput. Increasing the value size up to 64 bytes does not have impact on the throughout. The throughput degradation after 64 bytes (i.e., cache line size) is due to the use of the multiple cache lines in the Work Queue Element (WQEs). we performed an experiment with

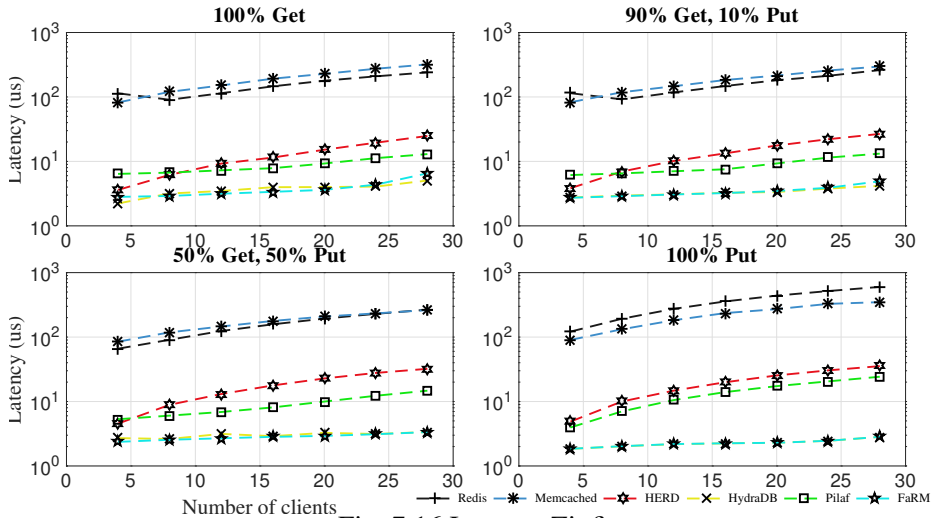

Fig. 7.16 Latency Zipfian.

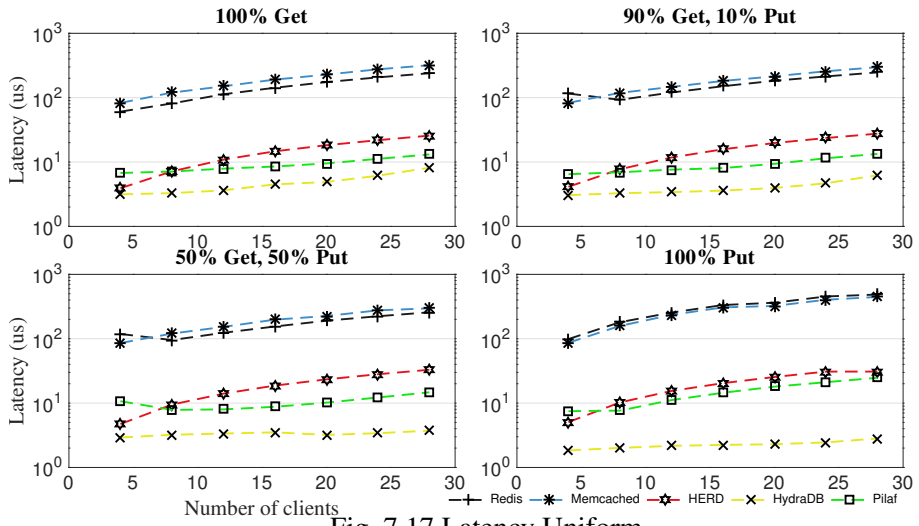

Fig. 7.17 Latency Uniform.

8 clients and 50% read-write ratio to measure the impact of varying the value size on the throughput. As Fig. 7.18 shows, the throughput of the systems reduce up to 1.7× by increasing the value size. Moreover, the HydraDB and FaRM are more sensitive to the size of the value due to inline `WRITE` in the experiment. The transition point when performance begins to drop depends on the NIC and the size of the value [44].

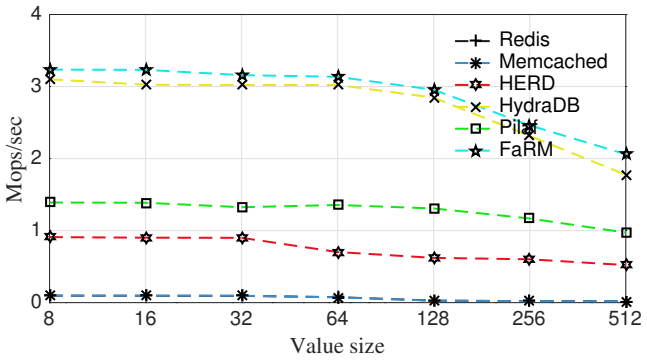

Fig. 7.18 Value size impact on the performance of the systems with 8 clients and 50% Get and Put.

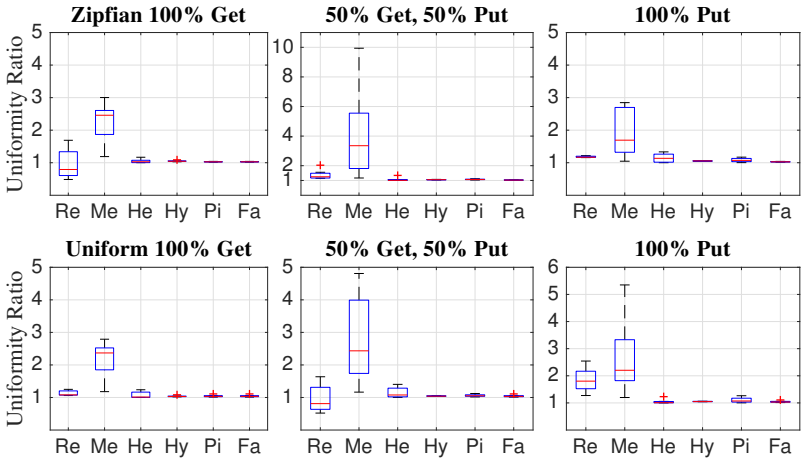

Fig. 7.19 Uniformity Ratio on 2 machines. Names are summarized to the first two letters.

7.3.4 Uniformity Ratio

Since the server copes with a large number of simultaneous access, satisfying the uniformity of the client requests is essential. The uniformity ratio indicates the maximum number of completed requests over the minimum number of completed requests among the clients. Closer ratio to 1 indicates the better uniformity and further ratio indicates the worse. We employ the uniformity ratio as it is an indicator of fairness [127]. As shown in Fig. 7.19, all RDMA-based systems have the uniformity ratio close to 1.

7.4 Kanzi Analysis

Our first major design decision in Kanzi is client-driven capability which allows clients to perform *Get* and *Put* transactions without intervention of server. This feature is important since server does not require to perform any polling operation to observe the new requests. Furthermore, Kanzi exploits a novel indirection table mechanism which drastically simplifies the problem of synchronization, since all clients observe the indirection table and RDMA atomic operations can be used to guarantee its consistency. In fact, the beauty of this design is that in case of update Kanzi clients only observe indirection table to retrieve the address of latest update, and it can be done by a single `READ` operation. The other novelty in Kanzi design is introducing lazy synchronization which is based on cache coherency of DMA and CPU on x86-based systems. This mechanism guarantee consistency on CHT when a new client join to the system. Moreover, latch-free *Get* transaction can help to achieve better performance. In the Kanzi design memory space efficiency is one of the concern to avoid network traffic.

To explore the performance characteristics of the design, the Kanzi prototype on the InfiniBand cluster is implemented. The specification of the experimental setup was described in Section 7.1. According to the design in order to evaluate the real performance at least three isolated nodes of cluster are required. One node to put the Kanzi shard and the two others are used to place the Kanzi clients. Two separated nodes for clients are required since putting all clients on one machine can bypass some part of the design and shows higher performance.

As described, workload is an important parameter which can influence on the performance. According to the design, if each client requests freely on every key on the shard, system can encounter a big overhead due to the ping pong on the request. For example, if two clients resides on two different machines are making request (i.e., *Put*) on the same key. If this key is hot key in both clients, then a ping pong loop will happen and the key is moving between these two machines. It happens because in each *Put* the key is moved to the client that issued the request. So, apart from the architecture a key management is required. However, this scenario will happen only when both clients have *Put* requests. Therefore, keys must be distributed among clients on different machines.

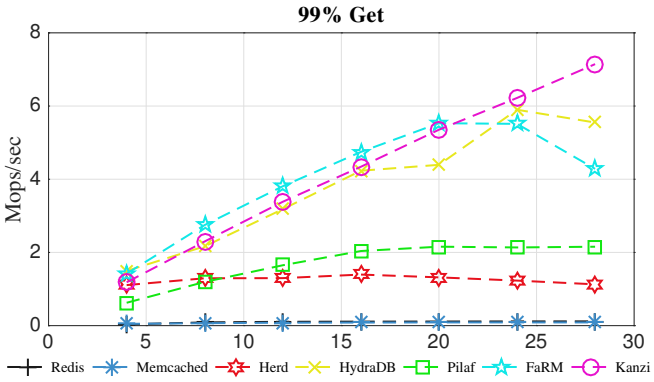

Fig. 7.20 Kanzi result.

According to the literature, most of the real-world workload are read-intensive. In spite of the fact that a list of experiment was designed to evaluate the performance of the Kanzi the limitation on the usage of the cluster did not let us to fully cover all the performance aspects and only the performance on the %99 percent read workload was experimentally presented.

Fig. 7.20 shows the performance of the Kanzi. As can be seen, Kanzi outperforms the state-of-the-art with higher number of clients. Although FaRM performs better in lower number of clients however this implementation of FaRM is for small message size which embed the value in the bucket. The performance achievement can be described by non-blocking *Get* operation in Kanzi as well as low amplification in *Get* operation. We believe that Kanzi outperforms much better than the state-of-the-art by increasing the *Put* ratio since the clients on the same node performs only one local access to read the data.

Chapter 8

Conclusion and Future work

The main motivation for the work presented in this thesis is rooted in exploiting RDMA in the design of in-memory key-value store. The focus was on the performance of the RDMA operations as well as analysing the state-of-the-art RDMA-enabled in-memory key-value stores. Then, proposing a novel RDMA-enabled in-memory key-value store based on the findings in the analysis. The performance analysis of RDMA operations helped us to have better insight on exploiting the RDMA operation in in-memory key-value stores. Towards this analysis, a good knowledge of design changes over time has been achieved to detect anomalies over systems.

In this thesis, motivations and main drivers of the NoSQL movement have been briefly described along with a classification and characterization of NoSQL databases. Then, the Remote Direct Memory Access (RDMA) in high-performance protocols have been discussed. Commonly used underlying data structure and concurrency control mechanisms in RDMA-enabled in-memory key-value store have been discussed.

Different performance challenging components in modern clusters such as memory, host bus communication, HCA memory, RDMA features, network communication, and application level issues have been investigated. One-of-a-kind comprehensive study of modern RDMA-based in-memory key-value systems including HydraDB [1], Pilaf [2], HERD [3], FaRM [4], and DrTM [5] as well as well-known legacy in-memory systems such as Memcached [6] and Redis [7] have been pre-

sented. Mentioned systems have been illustrated in an unified representation to show architectural differences along with strengths and weaknesses of each system.

To evaluate the RDMA-enabled in-memory key-value stores a set of unified experiment have been designed to measure throughput, latency, fairness, and uniformity ratio. These metrics are chosen since they are commonly used metrics in highly cited papers in this scope. To provide the reproducibility and interpretability of the experiments, the non-deterministic and deterministic parameters which can impact on the performance have been investigated. Redis 3.2.9 [7], Memcached 1.4.37 [6], and HERD [122] are employed from the original source. However, HydraDB, Pilaf, and FaRM are implemented from scratch since they are not publically available.

Finally, an RDMA-enabled in-memory key-value design, called Kanzi have been presented that allows concurrent reads and writes by clients. It supports *Get* and *Put* operations from remote clients. *Get* leverages RDMA read operations on the key-value store and *Put* exploits RDMA write and atomic operations.

To make concurrent operations possible, a memory management that provides explicit regions to clients to write and read in combination with a special memory reclamation method as well as concurrency control was proposed. The proposed memory management scheme allows the client to write to a region of memory not only in the server but also in the client address spaces. Kanzi also proposes a novel concurrency mechanism based on cache coherency called lazy synchronization. It exploits a central table to keep the latest update of each key and let clients to modify it with RDMA compare-and-swap (`CAS`) operations.

We plan to expand our evolution by analyzing other parameters presented in Section 7.3. Also, we intend to apply our approach in real-world use cases to further verify the results. Moreover, for consistency characteristics, we plan to present the formal verification of the proposed methods. As a future work, we plan also to perform additional evaluations on the key distribution impact on the performance. We will explore the impact of hash table load factors on the performance. Moreover, we intend to investigate how well Kanzi scales with multiple servers. Finally, we intend to perform a power-efficiency comparison against conventional key-value stores.

Examining large amounts of data can help to uncover hidden patterns and correlations. The area of this research will be particularly important as far as the role of big data and data science is growing in the society. Data Science helps humans

to make better decisions. Moreover, it can add value to all the business models to get the right information and make the right decisions. So, one need is accelerating data access. Many of the presented insights in this thesis can be used in the design of RDMA-enabled in-memory key-value store to enhance the performance of data access.

References

[1] Yandong Wang, Li Zhang, Jian Tan, Min Li, Yuqing Gao, Xavier Guerin, Xiaoqiao Meng, and Shicong Meng. HydraDB: A resilient RDMA-driven key-value middleware for in-memory cluster computing. In *SC'15*.

[2] Yifeng Geng Christopher Mitchell and Jinyang Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 103–114, San Jose, CA, 2013. USENIX.

[3] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using RDMA efficiently for key-value services. In *Special Interest Group on Data Communication*, volume 44, pages 295–306. ACM, 2014.

[4] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, pages 401–414, 2014.

[5] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast In-memory Transaction Processing Using RDMA and HTM. In *Symposium on Operating Systems Principles*, pages 87–104. ACM, 2015.

[6] http://memcached.org/. Accessed: 12-October-2018.

[7] Salvatore Sanfilippo. http://redis.io/. Accessed: 12-October-2018.

[8] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[9] http://www.hypertable.org/. Accessed: 12-October-2018.

[10] https://hbase.apache.org/. Accessed: 12-October-2018.

[11] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proceedings of the VLDB Endowment*, 2(1):922–933, 2009.

[12] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *Special interest Group in Operating Systems*, 44(2):35–40, 2010.

[13] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (xml). *World Wide Web Journal*, 2(4):27–66, 1997.

[14] http://www.json.org/. Accessed: 12-October-2018.

[15] https://www.mongodb.com. Accessed: 12-October-2018.

[16] http://couchdb.apache.org. Accessed: 12-October-2018.

[17] https://ravendb.net. Accessed: 12-October-2018.

[18] https://neo4j.com/. Accessed: 12-October-2018.

[19] http://titan.thinkaurelius.com/. Accessed: 12-October-2018.

[20] http://www.franz.com/agraph/allegrograph/. Accessed: 12-October-2018.

[21] https://www.objectivity.com/products/infinitegraph/. Accessed: 12-October-2018.

[22] http://infogrid.org/trac/. Accessed: 12-October-2018.

[23] http://basho.com/products/riak-kv. Accessed: 12-October-2018.

[24] http://www.project-voldemort.com/voldemort/. Accessed: 12-October-2018.

[25] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *Special interest Group in Operating Systems*, 41(6):205–220, 2007.

[26] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: A Memory-efficient, High-performance Key-value Store. In *Symposium on Operating Systems Principles*, pages 1–13, New York, NY, USA, 2011. ACM.

[27] Benjamin Cassell, Tyler Szepesi, Bernard Wong, Tim Brecht, Jonathan Ma, and Xiaoyi Liu. Nessie: A Decoupled, Client-Driven Key-Value Store Using RDMA. *IEEE Transactions on Parallel and Distributed Systems*, 28(12):3537–3552, 2017.

[28] Jilong Xue, Youshan Miao, Cheng Chen, Ming Wu, Lintao Zhang, and Lidong Zhou. Rpc considered harmful: Fast distributed deep learning on rdma. *arXiv preprint arXiv:1805.08430*, 2018.

[29] Yufei Ren, Xingbo Wu, Li Zhang, Yandong Wang, Wei Zhang, Zijun Wang, Michel Hack, and Song Jiang. irdma: Efficient use of rdma in distributed deep learning systems. In *High Performance Computing and Communications; International Conference on Smart City*, pages 231–238. IEEE, 2017.

[30] Chengfan Jia, Junnan Liu, Xu Jin, Han Lin, Hong An, Wenting Han, Zheng Wu, and Mengxian Chi. Improving the performance of distributed tensorflow with rdma. *International Journal of Parallel Programming*, pages 1–12, 2017.

[31] Brent Callaghan, Theresa Lingutla-Raj, Alex Chiu, Peter Staubach, and Omer Asad. Nfs over rdma. In *Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence: experience, lessons, implications*, pages 196–208. ACM, 2003.

[32] David Sidler, Zsolt István, and Gustavo Alonso. Low-latency TCP/IP stack for data center applications. In *Field Programmable Logic and Applications*, pages 1–4. IEEE, 2016.

[33] Ian Pratt and Keir Fraser. Arsenic: A user-accessible gigabit ethernet interface. In *Joint Conference of the IEEE Computer and Communications Societies*, volume 1, pages 67–76. IEEE, 2001.

[34] Thorsten Von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A user-level network interface for parallel and distributed computing. In *Special interest Group in Operating Systems*, volume 29, pages 40–53. ACM, 1995.

[35] Dave Dunning, Greg Regnier, Gary McAlpine, Don Cameron, Bill Shubert, Frank Berry, Anne Marie Merritt, Ed Gronke, and Chris Dodd. The virtual interface architecture. *IEEE micro*, 18(2):66–76, 1998.

[36] Nanette J Boden, Danny Cohen, Robert E Felderman, Alan E. Kulawik, Charles L Seitz, Jakov N Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE micro*, 15(1):29–36, 1995.

[37] Jon Beecroft, David Addison, David Hewson, Moray McLaren, Duncan Roweth, Fabrizio Petrini, and Jarek Nieplocha. QSNET/sup II: defining high-performance network design. *IEEE micro*, 25(4):34–47, 2005.

[38] http://infinibandta.org/index.php. Accessed: 12-October-2018.

[39] https://tools.ietf.org/html/rfc5040. Accessed: 12-October-2018.

[40] Mark S Birrittella, Mark Debbage, Ram Huggahalli, James Kunz, Tom Lovett, Todd Rimmer, Keith D Underwood, and Robert C Zak. Intel® omni-path architecture: Enabling scalable, high performance fabrics. In *High-Performance Interconnects*, pages 1–9. IEEE, 2015.

[41] Kevin Gildea, Rama Govindaraju, Donald Grice, Peter Hochschild, and Fu Chung Chang. Remote direct memory access system and method, August 30 2004. US Patent App. 10/929,943.

[42] https://www.mellanox.com/pdf/whitepapers/wp_roce_vs_iwarp.pdf. Accessed: 12-July-2018.

[43] https://www.openfabrics.org/. Accessed: 12-July-2018.

[44] Anuj Kalia Michael Kaminsky and David G Andersen. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference*, 2016.

[45] Allyn Romanow and Stephen Bailey. An overview of rdma over ip. In *Proceedings of the First International Workshop on Protocols for Fast Long-Distance Networks*, 2003.

[46] Claude Barthels, Gustavo Alonso, and Torsten Hoefler. Designing Databases for Future High-Performance Networks. *IEEE Data Eng. Bull.*, 40(1):15–26, 2017.

[47] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26(2), 2008.

[48] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling Memcache at Facebook. In *Networked Systems Design and Implementation*, volume 13, pages 385–398, 2013.

[49] Dong Dai, Xi Li, Chao Wang, Mingming Sun, and Xuehai Zhou. Sedna: A memory based key-value storage system for realtime processing in cloud. In *Cluster Computing Workshops*, pages 48–56. IEEE, 2012.

[50] Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, Antoine Atallah, Ralf Herbrich, Stuart Bowers, et al. Practical lessons from predicting clicks on ads at facebook. In *International Workshop on Data Mining for Online Advertising*, pages 1–9. ACM, 2014.

[51] Cheng Li, Yue Lu, Qiaozhu Mei, Dong Wang, and Sandeep Pandey. Click-through prediction for advertising in twitter timeline. In *International Conference on Knowledge Discovery and Data Mining*, pages 1959–1968. ACM, 2015.

[52] Yanxiang Huang, Bin Cui, Wenyu Zhang, Jie Jiang, and Ying Xu. Tencentrec: Real-time stream recommendation in practice. In *International Conference on Management of Data*, pages 227–238. ACM, 2015.

[53] Goetz Graefe. The five-minute rule twenty years later, and how flash memory changes the rules. In *International workshop on Data management on new hardware*. ACM, 2007.

[54] Prashanth Menon, Tilmann Rabl, Mohammad Sadoghi, and Hans-Arno Jacobsen. CaSSanDra: An SSD boosted key-value store. In *International Conference on Data Engineering*, pages 1162–1167. IEEE, 2014.

[55] Biplob Debnath, Sudipta Sengupta, and Jin Li. FlashStore: high throughput persistent key-value store. *Proceedings of the Very Large Database Endowment*, 3(2):1414–1425, 2010.

[56] https://www.facebook.com/note.php?note_id=388112370932. Accessed: 12-October-2018.

[57] Ashok Anand, Steven Kappes, Aditya Akella, and Suman Nath. Building cheap and large cams using bufferhash. *University of Wisconsin Madison Technical Report TR1651*, 2009.

[58] Diego Ongaro, Stephen M Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 29–41. ACM, 2011.

[59] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, 2014. USENIX Association.

[60] Jithin Jose, Hari Subramoni, Miao Luo, Minjia Zhang, Jian Huang, Md Wasiur Rahman, Nusrat S Islam, Xiangyong Ouyang, Hao Wang, Sayantan Sur, et al. Memcached design on high performance rdma capable interconnects. In *International Conference on Parallel Processing*, pages 743–752. IEEE, 2011.

[61] Charles Chen. An overview of cuckoo hashing.

[62] Donald Ervin Knuth. *The art of computer programming: fundamental algorithms*, volume 1. Addison Wesley Longman, 1997.

[63] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *European Symposium on Algorithms*, pages 121–133. Springer, 2001.

[64] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663. ACM, 1997.

[65] Yandong Wang, Xiaoqiao Meng, Li Zhang, and Jian Tan. C-hint: An effective and reliable cache management for rdma-accelerated key-value stores. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 23:1–23:13. ACM, 2014.

[66] Anuj Kalia, Michael Kaminsky, and David G Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs. In *Operating Systems Design and Implementation*, pages 185–201, 2016.

[67] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch hashing. In *International Symposium on Distributed Computing*, pages 350–364. Springer, 2008.

[68] Radhika Mittal, Alex Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting Network Support for RDMA.

[69] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 2016.

[70] Omid Shahmirzadi. High-performance communication primitives and data structures on message-passing manycores:broadcast and map. 2014.

[71] http://www.mellanox.com/page/multi_core_overview?mtag=multi_core_overview. Accessed: 12-October-2018.

[72] http://www.kalrayinc.com/. Accessed: 12-October-2018.

[73] David Slogsnat, Alexander Giese, Mondrian Nüssle, and Ulrich Brüning. An open-source HyperTransport core. *ACM Transactions on Reconfigurable Technology and Systems*, 1(3), 2008.

[74] Dimitrios Ziakas, Allen Baum, Robert A Maddox, and Robert J Safranek. Intel® quickpath interconnect architectural features supporting scalable system architectures. In *High Performance Interconnects*, pages 1–6. IEEE, 2010.

[75] http://pcisig.com/specifications/pciexpress/. Accessed: 12-October-2018.

[76] https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html. Accessed: 12-October-2018.

[77] Frank Mietke, Robert Rex, Robert Baumgartl, Torsten Mehlan, Torsten Hoefler, and Wolfgang Rehm. Analysis of the memory registration process in the mellanox infiniband software stack. *International European Conference on Parallel and Distributed Computing*, pages 124–133, 2006.

[78] Donghyuk Lee, Lavanya Subramanian, Rachata Ausavarungnirun, Jongmoo Choi, and Onur Mutlu. Decoupled direct memory access: Isolating cpu and io traffic by leveraging a dual-data-port dram. In *Parallel Architecture and Compilation*, pages 174–187. IEEE, 2015.

[79] https://cw.infinibandta.org/document/dl/7859. Accessed: 12-October-2018.

[80] Philip Werner Frey and Gustavo Alonso. Minimizing the hidden cost of RDMA. In *International Conference on Distributed Computing Systems*, pages 553–560. IEEE, 2009.

[81] Li Ou, Xubin He, and Jizhong Han. An efficient design for fast memory registration in RDMA. *Journal of Network and Computer Applications*, 32(3):642–651, 2009.

[82] Qian Liu and Robert D Russell. A performance study of infiniband fourteen data rate (fdr). In *Proceedings of the High Performance Computing Symposium*. Society for Computer Simulation International, 2014.

[83] Randal E Bryant, O'Hallaron David Richard, and O'Hallaron David Richard. *Computer Systems A Programmer's Perspective*, volume 2. Prentice Hall Upper Saddle River, 2003.

[84] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. Thread and Memory Placement on NUMA Systems: Asymmetry Matters. In *USENIX Annual Technical Conference*, pages 277–289, 2015.

[85] Tan Li, Yufei Ren, Dantong Yu, and Shudong Jin. Analysis of numa effects in modern multicore systems for the design of high-performance data transfer applications. *Future Generation Computer Systems*, 74:41–50, 2017.

[86] Yufei Ren, Tan Li, Dantong Yu, Shudong Jin, and Thomas Robertazzi. Design and performance evaluation of numa-aware rdma-based end-to-end data transfer systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013.

[87] Abdel-Hameed A Badawy, Aneesh Aggarwal, Donald Yeung, and Chau-Wen Tseng. Evaluating the impact of memory system performance on software prefetching and locality optimizations. In *Proceedings of the 15th international conference on Supercomputing*, pages 486–500. ACM, 2001.

[88] David Callahan, Ken Kennedy, and Allan Porterfield. Software Prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52. ACM, 1991.

[89] http://www.amd.com/en/technologies/sense-mi. Accessed: 12-October-2018.

[90] https://newsroom.intel.com/news-releases/pci-sig-introduces-pci-express-formerly-3gio-high-speed-serial-interconnect-specification/. Accessed: 12-October-2018.

[91] Robert Bruce Thompson and Barbara Fritchman Thompson. *PC Hardware in a Nutshell, 3rd Edition*. O'Reilly & Associates, Inc., 2003.

[92] Ranjit Noronha and Dhabaleswar K Panda. Can high performance software dsm systems designed with infiniband features benefit from pci-express? In *Cluster, Cloud and Grid computing*, volume 2, pages 945–952. IEEE, 2005.

[93] Jiuxing Liu, Amith Mamidala, Abhinav Vishnu, and Dhabaleswar K Panda. Performance evaluation of infiniband with pci express. In *High Performance Interconnects, 2004. Proceedings. 12th Annual IEEE Symposium on*, pages 13–19. IEEE, 2004.

[94] https://community.mellanox.com/docs/doc-2491. Accessed: 12-October-2018.

[95] https://www.xilinx.com/support/documentation/white_papers/wp350.pdf. Accessed: 12-October-2018.

[96] https://community.mellanox.com/docs/doc-2496. Accessed: 12-October-2018.

[97] https://community.mellanox.com/docs/doc-2123. Accessed: 12-October-2018.

[98] Mario Flajslik and Mendel Rosenblum. Network Interface Design for Low Latency Request-Response Protocols. In *USENIX Annual Technical Conference*, pages 333–346, 2013.

[99] http://www.mellanox.com/related-docs/user_manuals/ ethernet_adapters_programming_manual.pdf. Accessed: 12-October-2018.

[100] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W Moore. Understanding pcie performance for end host networking. pages 327–341. ACM, 2018.

[101] Sayantan Sur, Abhinav Vishnu, H-W Jin, DK Panda, and W Huang. Can memory-less network adapters benefit next-generation infiniband systems? In *High Performance Interconnects, 2005. Proceedings. 13th Symposium on*, pages 45–50. IEEE, 2005.

[102] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Symposium on Operating Systems Principles*, pages 54–70. ACM, 2015.

[103] https://cw.infinibandta.org/document/dl/7146. Accessed: 12-October-2018.

[104] Tom Shanley. *Infiniband Network Architecture*. Addison-Wesley Professional, 2003.

[105] Patrick MacArthur and Robert D Russell. A Performance Study to Guide RDMA Programming Decisions. In *International Conference on High Performance Computing and Communication & International Conference on Embedded Software and Systems*, pages 778–785. IEEE, 2012.

[106] http://www.mellanox.com/related-docs/prod_software /rdma_aware_programming_user_manual.pdf. Accessed: 12-October-2018.

[107] Patrick MacArthur, Qian Liu, Robert D Russell, Fabrice Mizero, Malathi Veeraraghavan, and John M Dennis. An integrated tutorial on InfiniBand, Verbs and MPI. *IEEE Communications Surveys & Tutorials*, 2017.

[108] Jerome Vienne, Jitong Chen, Md Wasi-Ur-Rahman, Nusrat S Islam, Hari Subramoni, and Dhabaleswar K Panda. Performance analysis and evaluation of infiniband fdr and 40gige roce on hpc and cloud computing systems. In *High-Performance Interconnects*, pages 48–55. IEEE, 2012.

[109] Motti Beck and Michael Kagan. Performance evaluation of the rdma over ethernet (roce) standard in enterprise data centers infrastructure. In *Proceedings of the 3rd Workshop on Data Center - Converged and Virtual Ethernet Switching*, pages 9–15. International Teletraffic Congress, 2011.

[110] Hari Subramoni, Ping Lai, Miao Luo, and Dhabaleswar K Panda. RDMA over Ethernet—A preliminary study. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–9. IEEE, 2009.

[111] Luke Anthony Kachelmeier, Faith Virginia Van Wig, and Kari Natania Erickson. Comparison of high performance network options: Edr infiniband vs. 100gb rdma capable ethernet. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2016.

[112] Matthew J Koop, Wei Huang, Karthik Gopalakrishnan, and Dhabaleswar K Panda. Performance analysis and evaluation of pcie 2.0 and quad-data rate infiniband. In *High Performance Interconnects*, pages 85–92. IEEE, 2008.

[113] Jiuxing Liu, Amith Mamidala, Abhinav Vishnu, and Dhabaleswar K Panda. Evaluating infiniband performance with pci express. *IEEE Micro*, 25(1):20–29, 2005.

[114] Infiniband pci pcie. Accessed: 12-October-2018.

[115] https://www.intel.in/content/dam/www/public/us/en/documents/product-briefs/xeon-processor-d-brief.pdf. Accessed: 12-October-2018.

[116] https://www.nersc.gov/assets/uploads/knl-isc-2015-workshop-keynote.pdf. Accessed: 12-October-2018.

[117] Dennis Dalessandro and Pete Wyckoff. Memory management strategies for data serving with RDMA. In *High-Performance Interconnects*, pages 135–142. IEEE, 2007.

[118] Yiwen Zhang, Juncheng Gu, Youngmoon Lee, Mosharaf Chowdhury, and Kang G Shin. Performance Isolation Anomalies in RDMA. In *Workshop on Kernel-Bypass Networks*, pages 43–48, 2017.

[119] Mohammad Sadoghi, Kenneth A. Ross, Mustafa Canim, and Bishwaranjan Bhattacharjee. Exploiting SSDs in operational multiversion databases. *Very Large Database*, 25(5):651–672, 2016.

[120] Mohammad Sadoghi, Mustafa Canim, Bishwaranjan Bhattacharjee, Fabian Nagel, and Kenneth A. Ross. Reducing Database Locking Contention Through Multi-version Concurrency. *Very Large Database*.

[121] Mohammad Sadoghi, Mustafa Canim, Bishwaranjan Bhattacharjee, Fabian Nagel, and Kenneth A Ross. Reducing database locking contention through multi-version concurrency. *Proceedings of the Very Large Database Endowment*, 7(13):1331–1342, 2014.

[122] https://github.com/efficient/rdma_bench. Accessed: 12-October-2018.

[123] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 126–134. IEEE, 1999.

[124] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. An Analysis of Facebook Photo Caching. In *Symposium on Operating Systems Principles*, pages 167–181. ACM, 2013.

[125] Bin Fan, David G Andersen, and Michael Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Symposium on Networked Systems Design and Implementation*, volume 13, pages 371–384, 2013.

[126] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[127] Darko Petrović, Thomas Ropars, and André Schiper. Leveraging Hardware Message Passing for Efficient Thread Synchronization. In *Symposium on Principles and Practice of Parallel Programming*, pages 143–154. ACM, 2014.

Appendix A

Doctoral Period's Publications

This doctoral thesis is based on the following publications:

1. [URL] Hemmatpour, M., Montrucchio, B., Rebaudengo, M., & Sadoghi, M. Kanzi: A distributed, in-memory key-value store. In Proceedings of the Posters and Demos Session of the Middleware Conference, pp. 3-4, 2016.

2. [URL] Hemmatpour, M., Montrucchio, B., Rebaudengo, M. Communicating Efficiently on Cluster-Based Remote Direct Memory Access (RDMA) over InfiniBand Protocol. In Applied sciences vol. 8, no. 11, 2018.

In addition, an extended version of [1] has been submitted to the IEEE Transactions on Knowledge and Data Engineering (TKDE). Furthermore, the list of publications in the PhD period is as follows:

1. [URL] Hemmatpour, M., Ghazivakili, M., Montrucchio, B., & Rebaudengo, M. DIIG: A distributed industrial IoT gateway. In IEEE Annual Conference Computer Software and Applications Conference vol. 1, pp. 755-759, 2017

2. [URL] Hemmatpour, M., Ferrero, R., Montrucchio, B., & Rebaudengo, M. A Neural Network Model Based on Co-occurrence Matrix for Fall Prediction. In Proceedings of the Conference on Wireless Mobile Communication and Healthcare , vol. 192, p. 241, 2016.

3. [URL] Hemmatpour, M., Ferrero, R., Montrucchio, B., & Rebaudengo, M. Eigenwalk: a novel feature for walk classification and fall prediction. In Proceedings of the Conference on Body Area Networks, pp. 86-90, 2016.

4. [URL] Hemmatpour, M., Ferrero, R., Montrucchio, B., & Rebaudengo, M. A baseline walking dataset exploiting accelerometer and gyroscope for fall prediction and prevention systems. In Proceedings of the Conference on Body Area Networks, pp. 81-85, 2016.

5. [URL] Ferrero, R., Gandino, F., Hemmatpour, M., Montrucchio, B., & Rebaudengo, M. Exploiting accelerometers to estimate displacement. In IEEE Mediterranean Conference on Embedded Computing, pp. 206-210, IEEE.

6. [URL] Hemmatpour, M., Ferrero, R., Gandino, F., Montrucchio, B., & Rebaudengo, M. Nonlinear Predictive Threshold Model for Real-Time Abnormal Gait Detection. Journal of healthcare engineering, vol. 2018, pp. 1-9, 2018.

7. [URL] Hemmatpour, M., Ferrero, R., Gandino, F., Montrucchio, B., & Rebaudengo, M. Data Reduction Techniques for Near Real-Time Decision Making in Fall Prediction Systems. In Big Data Management and the Internet of Things for Improved Health Systems, pp. 52-64, 2018.

8. [URL] Hemmatpour, M., Ferrero, R., Gandino, F., Montrucchio, B., & Rebaudengo, M. Cost Evaluation of Synchronization Algorithms for Multicore Architectures. In Encyclopedia of Information Science and Technology, Fourth Edition, pp. 3989-4003, 2018.

9. [URL] Ferrero, R., Gandino, F., Hemmatpour, M., Montrucchio, B., & Rebaudengo, M. Urban dust monitoring from ground level to last floor. In IEEE Conference on Mobile Computing and Ubiquitous Network, pp. 1-4, 2017.

10. [URL] Hemmatpour, M., Karimshoushtari, M., Ferrero, R., Montrucchio, B., Rebaudengo, M., & Novara, C. Polynomial classification model for real-time fall prediction system. In IEEE Annual Computer Software and Applications Conference, pp. 973-978, 2017.

11. [URL] Hemmatpour, M., Ferrero, R., Gandino, F., Montrucchio, B., & Rebaudengo, M. Internet of Things for fall prediction and prevention. Journal of Computational Methods in Sciences and Engineering, vol. 18, no. 2, pp. 511-518, 2018.